Notes on Problem Solving

Gyubeom Edward Im*

Contents

*blog: alida.tistory.com, email: criterion.im@gmail.com

NOMENCLATURE

- V : 노드(Node, Vertex)의 개수
- E : 간선(Edge)의 개수
- n : 원소의 개수 (또는 N)
- $\log n$: 밑이 2인 $\log_2 n$을 간략하게 표기
- 노드 = 정점
- 간선 = 엣지

1 Data structure

1.1 Array

배열(Array)는 메모리 상에 원소를 연속하게 배치한 자료구조를 말한다. C++에서는 배열의 원소가 한 번 정해지면 (`int A[10]`) 이후 크기를 변경하는 것이 일반적으로 불가능하다.

- C++ Skeleton Code (Pure) #1

```cpp
#include <cstdio>

int A[10] = {10, 20, 30};
int len = 3;

void insert(int idx, int num) {
  for(int i=len; i>idx; i--) A[i] = A[i-1];
  A[idx] = num;
  len++;
}

void erase(int idx) {
  len--;
  for(int i=idx; i<len; i++) A[i] = A[i+1];
}

void Print() {
  for(int i=0; i<len; i++) printf("%d ", A[i]);
  puts("");
}


int main() {
  printf("insert test\n");
  insert(3, 40); Print(); // 10 20 30 40
  insert(1, 50); Print(); // 10 50 20 30 40
  insert(0, 15); Print(); // 15 10 50 20 30 40

  printf("erase test\n");
  erase(4); Print(); // 15 10 50 20 40
  erase(1); Print(); // 15 50 20 40
  erase(3); Print(); // 15 50 20
}
```

- C++ Skeleton Code (Pure) #2

```cpp
#include <bits/stdc++.h>
using namespace std;

int* A;
int len = 0;     // 실제 데이터가 들어있는 크기
int capacity = 0; // 삽입 가능한 최대 크기

void init(){
```

```
    A = new int[1];
    capacity = 1;
  }

  void expand(){
    int* tmp = new int[2*capacity];
    for(int i=0;i<len;i++)
    tmp[i] = A[i];
    delete[] A;
    A = tmp;
    capacity = 2*capacity;
  }

  void insert(int idx, int num) {
    if(len == capacity) expand();

    for(int i=len; i>idx; i--) A[i] = A[i-1];
    A[idx] = num;
    len++;
  }

  void erase(int idx) {
    len--;
    for(int i=idx; i<len; i++) A[i] = A[i+1];
  }

  void Print() {
    for(int i=0; i<len; i++) printf("%d ", A[i]);
    puts("");
  }

  int main() {
    init();

    printf("insert test\n");
    insert(0, 10); Print(); // 10, len = 1, capacity = 1
    insert(0, 30); Print(); // 30 10, len = 2, capacity = 2
    insert(1, 20); Print(); // 30 20 10, len = 3, capacity = 4
    insert(3, 40); Print(); // 30 20 10 40, len = 4, capacity = 4
    insert(1, 50); Print(); // 30 50 20 10 40, len = 5, capacity = 8
    insert(0, 15); Print(); // 15 30 50 20 10 40, len = 6, capacity = 8
  }
```

동적 배열이 필요한 경우 매 번 길이가 capacity에 도달할 때마다 배열의 길이를 2배씩 증가시켜주면 삽입 연산의 시간복잡도가 Amortized $O(1)$이 된다. (사실 상 $O(1)$과 동일한 시간복잡도) (BaaaaaaaarkingDog 님 동적 배열 참조)

1.2 std::vector

STL의 벡터(vector) 컨테이너를 사용하면 배열과 유사하지만 크기를 가변적으로 사용할 수 있다. 벡터는 가변적인 배열의 역할을 수행하기 때문에 배열을 사용하는 거의 모든 C++ 코드에서 광범위하게 사용된다.

- operator= : 깊은 복사(deep copy)가 발생한다.
- 시간복잡도
  - `push_back` : 원소를 끝에 추가 $O(1)$
  - `pop_back` : 마지막 원소 제거 $O(1)$
  - `[], at` : 임의의 위치에 원소 확인/변경: $O(1)$
  - `insert, erase` : 임의의 위치에 원소 추가/제거 $O(n)$
- C++ Skeleton Code (w/ STL)

```
#include <cstdio>
#include <vector>
```

```cpp
using namespace std;

vector<int> A;

void Print() {
  for(int i=0; i<A.size(); i++) printf("%d ", A[i]);
  puts("");
}

int main() {
  A.push_back(10), A.push_back(20), A.push_back(30);

  printf("insert test\n");
  A.insert(A.begin()+3, 40); Print(); // 10 20 30 40
  A.insert(A.begin()+1, 50); Print(); // 10 50 20 30 40
  A.insert(A.begin()+0, 15); Print(); // 15 10 50 20 30 40

  printf("erase test\n");
  A.erase(A.begin()+4); Print(); // 15 10 50 20 40
  A.erase(A.begin()+1); Print(); // 15 50 20 40
  A.erase(A.begin()+3); Print(); // 15 50 20
}
```

1.3 Linked list

연결 리스트(Linked List)는 노드들이 포인터로 연결되어 있는 선형 자료구조이다. 각 노드는 데이터와 다음 노드를 가리키는 포인터로 구성되어 있으며, 데이터의 동적 추가 및 삭제가 용이하다. 연결하는 방법 및 개수에 따라 단일 연결 리스트(single linked list), 이중 연결 리스트(double linked list), 원형 연결 리스트(circular linked list)로 구분할 수 있다.

- 배열과 달리 임의의 원소로 가기 위해서는 첫번째 원소부터 순서대로 방문해야 한다.
- 배열과 달리 메모리 상의 배치가 불연속적이다.
- 다음 원소의 주소값을 가지고 있어야 하므로 64비트(=8바이트) 컴퓨터 기준 $8N$ 바이트만큼 추가적인 메모리가 필요하다. (배열 대비 오버헤드가 있음)
- C++ Skeleton Code (Pure)

```cpp
#include <bits/stdc++.h>
using namespace std;

const int LM = 1000005;
int dat[LM], pre[LM], nxt[LM];
int unused = 1;

void insert(int addr, int num) {
  dat[unused] = num;
  pre[unused] = addr;
  nxt[unused] = nxt[addr];
  if(nxt[addr] != -1) pre[nxt[addr]] = unused;
  nxt[addr] = unused;
  unused++;
}

void erase(int addr){
  nxt[pre[addr]] = nxt[addr];
  if(nxt[addr] != -1) pre[nxt[addr]] = pre[addr];
}

void traverse(){
  int cur = nxt[0];
  while(cur != -1){
    printf("%d ", dat[cur]);
    cur = nxt[cur];
  }
  puts("");
```

```
}

int main(int argc, char **argv){
  fill(pre, pre+LM, -1);
  fill(nxt, nxt+LM, -1);

  printf("****** insert_test *****\n");
  insert(0, 10); // 10(address=1)
  traverse();
  insert(0, 30); // 30(address=2) 10
  traverse();
  insert(2, 40); // 30 40(address=3) 10
  traverse();
  insert(1, 20); // 30 40 10 20(address=4)
  traverse();
  insert(4, 70); // 30 40 10 20 70(address=5)
  traverse();

  printf("****** erase_test *****\n");
  erase(1); // 30 40 20 70
  traverse();
  erase(2); // 40 20 70
  traverse();
  erase(4); // 40 70
  traverse();
  erase(5); // 40
  traverse();
}
```

1.4 std::list

STL의 list 컨테이너는 이중 연결 리스트(double linked list)이다.

- 시간복잡도
  - k번째 원소를 확인/변경 $O(k)$
  - 임의의 위치에 원소를 추가/제거 $O(1)$
- C++ Skeleton Code (w/ STL)

```
#include <cstdio>
#include <list>
using namespace std;

list<int> L;

void traverse() {
  for(auto l : L) printf("%d ", l);
  puts("");
}

int main() {
  L = {1, 2};        // 1 2
  auto t = L.begin(); // t는 1을 가리킴

  L.push_front(10); traverse(); // 10 1 2
  L.push_back(5); traverse(); // 10 1 2 5
  L.insert(t, 6); traverse(); // 10 6 1 2 5
  t = L.erase(t);            // 1 제거, t가 가리키는 값은 2
  printf("%d", *t);          // 2
}
```

1.5 Stack

스택(Stack)은 마지막에 삽입된 요소가 가장 먼저 삭제되는 LIFO(Last In, First Out) 방식의 선형 자료구조이다. 이는 함수의 호출 스택 처리, 수식의 괄호 검사 등에 활용된다. 스택은 제일 상단이 아닌 나머지

원소들의 확인과 변경이 원칙적으로 불가능한 특징이 있다.

- 시간복잡도
  - 원소 추가/제거: $O(1)$ 제일 상단의 원소 확인: $O(1)$
- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
constexpr int LM = 1000005;

struct Stack{
  int stack[LM];
  int pos=0;

  void push(int x) { stack[pos++] = x; }
  void pop() { pos--; }
  bool empty() { return (pos==0); }
  int top() { return stack[pos-1]; }
} S;

int main() {
  S.push(5); S.push(4); S.push(3);
  printf("%d\n", S.top()); // 3
  S.pop(); S.pop();
  printf("%d\n", S.top()); // 5
  S.push(10); S.push(12);
  printf("%d\n", S.top()); // 12
  S.pop();
  printf("%d\n", S.top()); // 10
}
```

- C++ Skeleton Code (w/ STL)

```cpp
#include <cstdio>
#include <stack>
using namespace std;

stack<int> S;

int main() {
  S.push(5); S.push(4); S.push(3);
  printf("%d\n", S.top()); // 3
  S.pop(); S.pop();
  printf("%d\n", S.top()); // 5
  S.push(10); S.push(12);
  printf("%d\n", S.top()); // 12
  S.pop();
  printf("%d\n", S.top()); // 10
}
```

1.6 Queue

큐(Queue)는 처음 삽입된 요소가 가장 먼저 삭제되는 FIFO(First In, First Out) 방식의 선형 자료구조이다. 운영체제의 작업 스케줄링, 네트워크 요청 처리 등에 사용된다. 큐는 제일 앞/뒤가 아닌 나머지 원소들의 확인과 변경이 원칙적으로 불가능한 특징이 있다.

- 시간복잡도
  - 원소 추가/제거: $O(1)$
  - 제일 앞/뒤의 원소 확인: $O(1)$
- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
constexpr int LM = 1000005;
```

```

struct Queue{
  int fr=0, re=0;
  int que[LM];

  void push(int x) { que[re++] = x; }
  void pop() { fr++; }
  int front() { return que[fr]; }
  int back() { return que[re-1]; }
} Q;

int main() {
  Q.push(10); Q.push(20); Q.push(30);
  printf("%d %d\n", Q.front(), Q.back()); // 10 30
  Q.pop(); Q.pop();
  Q.push(15); Q.push(25);
  printf("%d %d\n", Q.front(), Q.back()); // 30 25
}
```

- C++ Skeleton Code (w/ STL)

```
#include <cstdio>
#include <queue>
using namespace std;

queue<int> Q;

int main() {
  Q.push(10); Q.push(20); Q.push(30);
  printf("%d %d\n", Q.front(), Q.back()); // 10 30
  Q.pop(); Q.pop();
  Q.push(15); Q.push(25);
  printf("%d %d\n", Q.front(), Q.back()); // 30 25
}
```

1.7 Deque

덱(Deque)은 양쪽 끝에서 삽입과 삭제가 가능한 자료구조로, 스택과 큐의 특성을 모두 갖는다. 더 유연한 데이터 처리가 가능해 다양한 알고리즘 구현에 유용하다.

- 시간복잡도
  - 원소 추가/제거: $O(1)$
  - 제일 앞/뒤의 원소 확인: $O(1)$
- 제일 앞/뒤가 아닌 나머지 원소들의 확인/변경은 원칙적으로 불가능 (하지만 STL deque는 가능)
- C++ Skeleton Code (Pure)

```
#include <cstdio>
constexpr int LM = 1000005;

int deque[2*LM+1];
int fr=LM, re=LM;

void push_front(int x){ deque[--fr] = x; }
void push_back(int x){ deque[re++] = x; }
void pop_front(){ fr++; }
void pop_back(){ re--; }
int front(){ return deque[fr]; }
int back(){ return deque[re-1]; }

int main() {
  push_back(30); // [30]
  printf("%d ", front()); printf("%d\n", back()); // 30 30
  push_front(25); // [25 30]
  push_back(12); printf("%d\n", back()); // 12
```

```
  push_back(62); // [25 30 12 62]
  pop_front(); // [30 12 62]
  printf("%d\n", front()); // 30
  pop_front(); // [12 62]
  printf("%d\n", back()); // 62
}
```

- C++ Skeleton Code (w/ STL)

```
#include <cstdio>
#include <deque>
using namespace std;

deque<int> DQ;

int main() {
  DQ.push_back(30); // [30]
  printf("%d ", DQ.front()); printf("%d\n", DQ.back()); // 30 30
  DQ.push_front(25); // [25 30]
  DQ.push_back(12); printf("%d\n", DQ.back()); // 12
  DQ.push_back(62); // [25 30 12 62]
  DQ.pop_front(); // [30 12 62]
  printf("%d\n", DQ.front()); // 30
  DQ.pop_front(); // [12 62]
  printf("%d\n", DQ.back()); // 62
}
```

1.8 Heap

힙(Heap)은 완전 이진 트리를 기반으로 하는 자료구조로, 각 노드의 값이 자식 노드보다 크거나 같은(최대 힙) 또는 작거나 같은(최소 힙)을 만족한다. 우선순위 큐의 구현에 주로 사용된다.

- C++ Skeleton Code (Pure) (Max heap)

```
#include <cstdio>

constexpr int NUM = 10;
constexpr int LM = 105;

int A[] = { 5,2,7,1,3,8,4,6,10,9 };

int heap[LM];
int hn = 0;

void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

void push(int nd) {
  heap[++hn] = nd;

  for (int c = hn; c > 1; c /= 2) {
    if (heap[c] > heap[c / 2]) {
      swap(heap[c], heap[c / 2]);
    }
    else {
      break;
    }
  }
}

void pop() {
  swap(heap[1], heap[hn--]);

  for (int c = 2; c <= hn; c *= 2) {
    if (c < hn && heap[c + 1] > heap[c]) {
```

```
      c++;
    }
    if (heap[c] > heap[c / 2]) {
      swap(heap[c], heap[c / 2]);
    }
    else {
      break;
    }
  }
}

int main() {
  for(int i = 0; i < NUM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  printf("\n");

  // max heap push
  for(int i = 0; i < NUM; i++) { push(A[i]); }

  for(int i = 1; i <= hn; i++) { printf("%d ", heap[i]); } // 10 9 7 6 8 5 4 1 3 2
  printf("\n");

  // max heap pop
  while (hn > 1) { pop();  }

  for(int i = 1; i <= NUM; i++) printf("%d ", heap[i]); // 1 2 3 4 5 6 7 8 9 10
  printf("\n");
}
```

- C++ Skeleton Code (Pure) (Min heap)

```
#include <cstdio>

constexpr int NUM = 10;
constexpr int LM = 105;

int A[] = { 5,2,7,1,3,8,4,6,10,9 };

int heap[LM];
int hn = 0;

void swap(int &a, int &b) {
  int t = a;
  a = b;
  b = t;
}

void push(int nd) {
  heap[++hn] = nd;

  for (int c = hn; c > 1; c /= 2) {
    if (heap[c] < heap[c / 2]) { // 변경: 최소 힙 조건
      swap(heap[c], heap[c / 2]);
    }
    else {
      break;
    }
  }
}

void pop() {
  swap(heap[1], heap[hn--]);

  for (int c = 2; c <= hn; c *= 2) {
    if (c < hn && heap[c + 1] < heap[c]) { // 변경: 최소 힙 조건
      c++;
    }
    if (heap[c] < heap[c / 2]) { // 변경: 최소 힙 조건
      swap(heap[c], heap[c / 2]);
    }
    else {
```

```
      break;
    }
  }
}

int main() {
  for(int i = 0; i < NUM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  printf("\n");

  // min heap push
  for(int i = 0; i < NUM; i++) { push(A[i]); }

  for(int i = 1; i <= hn; i++) { printf("%d ", heap[i]); } // 최소 힙 구조 확인
  printf("\n");

  // min heap pop
  while (hn > 1) { pop();  }

  for(int i = 1; i <= NUM; i++) printf("%d ", heap[i]); // 10 9 8 7 6 5 4 3 2 1
  printf("\n");
}
```

1.9 Priority queue

우선순위 큐(Priority Queue)는 원소들이 우선순위에 따라 정렬되며, 우선순위가 가장 높은 원소가 가장 먼저 삭제된다. 힙(Heap)을 사용하여 구현할 수 있으며, 다익스트라 같은 그래프 알고리즘에서 중요하게 사용된다.

- C++ Skeleton Code (Pure) (Max heap)

```
#include <cstdio>

#define LM 1000

class PriorityQueue {
  private:
  int heap[LM];
  int hn;

  void swap(int &a, int &b) {
    int t = a; a = b; b = t;
  }

  public:
  PriorityQueue() : hn(0) {}

  void push(int nd) {
    heap[++hn] = nd;
    for (int c = hn; c > 1; c /= 2) {
      if (heap[c] > heap[c / 2]) {
        swap(heap[c], heap[c / 2]);
      } else {
        break;
      }
    }
  }

  void pop() {
    if (hn == 0) return;
    swap(heap[1], heap[hn--]);
    for (int c = 2; c <= hn; c *= 2) {
      if (c < hn && heap[c + 1] > heap[c]) {
        c++;
      }
      if (heap[c] > heap[c / 2]) {
        swap(heap[c], heap[c / 2]);
      } else {
        break;
```

```
      }
    }
  }

  int top() {
    return hn > 0 ? heap[1] : -1;
  }

  bool empty() {
    return hn == 0;
  }
};

int main() {
  // Max heap
  PriorityQueue pq;
  pq.push(5);  pq.push(2);  pq.push(7); pq.push(1);  pq.push(3);
  pq.push(8);  pq.push(4);  pq.push(6);  pq.push(10); pq.push(9); // 5 2 7 1 3 8 4 6 10 9

  printf("%d ", pq.top());        // 10
  pq.pop(); printf("%d ", pq.top()); // 9
  pq.pop(); printf("%d ", pq.top()); // 8
  pq.pop(); printf("%d ", pq.top()); // 7
  pq.pop(); printf("%d ", pq.top()); // 6
  pq.pop(); printf("%d ", pq.top()); // 5
  pq.pop(); printf("%d ", pq.top()); // 4
  pq.pop(); printf("%d ", pq.top()); // 3
  pq.pop(); printf("%d ", pq.top()); // 2
  pq.pop(); printf("%d ", pq.top()); // 1
  printf("\n");
}
```

- C++ Skeleton Code (w/ STL) (Max heap)

```
#include <cstdio>
#include <queue>

#define LM 1000

int main() {
  // Max heap
  std::priority_queue<int> pq;
  pq.push(5);  pq.push(2);  pq.push(7); pq.push(1);  pq.push(3);
  pq.push(8);  pq.push(4);  pq.push(6);  pq.push(10); pq.push(9); // 5 2 7 1 3 8 4 6 10 9

  printf("%d ", pq.top());        // 10
  pq.pop(); printf("%d ", pq.top()); // 9
  pq.pop(); printf("%d ", pq.top()); // 8
  pq.pop(); printf("%d ", pq.top()); // 7
  pq.pop(); printf("%d ", pq.top()); // 6
  pq.pop(); printf("%d ", pq.top()); // 5
  pq.pop(); printf("%d ", pq.top()); // 4
  pq.pop(); printf("%d ", pq.top()); // 3
  pq.pop(); printf("%d ", pq.top()); // 2
  pq.pop(); printf("%d ", pq.top()); // 1
  printf("\n");
}
```

- C++ Skeleton Code (w/ STL) (Min heap)

```
#include <cstdio>
#include <vector>
#include <queue>

#define LM 1000

int main() {
  // Min heap
  std::priority_queue<int, std::vector<int>, std::greater<int>> pq;
```

```
  pq.push(5);  pq.push(2);  pq.push(7); pq.push(1);  pq.push(3);
  pq.push(8);  pq.push(4);  pq.push(6);  pq.push(10); pq.push(9); // 5 2 7 1 3 8 4 6 10 9

  printf("%d ", pq.top());        // 1
  pq.pop(); printf("%d ", pq.top()); // 2
  pq.pop(); printf("%d ", pq.top()); // 3
  pq.pop(); printf("%d ", pq.top()); // 4
  pq.pop(); printf("%d ", pq.top()); // 5
  pq.pop(); printf("%d ", pq.top()); // 6
  pq.pop(); printf("%d ", pq.top()); // 7
  pq.pop(); printf("%d ", pq.top()); // 8
  pq.pop(); printf("%d ", pq.top()); // 9
  pq.pop(); printf("%d ", pq.top()); // 10
  printf("\n");
}
```

1.10 Hash

해시(Hash)는 키-값 쌍을 저장하는데 사용되며, 해시 함수를 통해 키를 해시 테이블의 주소로 변환하여 데이터를 효율적으로 조회, 삽입, 삭제할 수 있다. 빠른 데이터 검색에 유리하다.

- C++ Skeleton Code (Pure) (Chaining)

```
// Hash using chaining : https://blog.encrypted.gg/1009
#include <bits/stdc++.h>
using namespace std;

const int M = 1'000'003;
const int a = 1'000;
const int MX = 500'005; //  - 128) * 64 + (‘ - 128) 뀀 - 128) * 64 + (‘ - 128) 뀰
      - 128) * 64 + (‘ - 128) 뀀 - 128) * 64 + (‘ - 128) 뀀
      - 128) * 64 + (‘ - 128) 퀙 - 128) * 64 + (‘ - 128) 뀀

int head[M];
int pre[MX], nxt[MX];
string key[MX];
int val[MX];
int unused = 0;

int myHash(string& s){
  int h=0;
  for(auto x : s)
  h = (h*a + x) % M;
  return h;
}

int find(string k) {
  int h = myHash(k);
  int idx = head[h];
  while(idx != -1) {
    if(key[idx] == k) return idx;
    idx = nxt[idx];
  }
  return -1;
}

void insert(string k, int v) {
  int idx = find(k);
  if(idx != -1) {
    val[idx] = v;
    return;
  }
  int h = myHash(k);
  key[unused] = k;
  val[unused] = v;
  if(head[h] != -1) {
    nxt[unused] = head[h];
    pre[head[h]] = unused;
  }
```

```
    head[h] = unused;
    unused++;
  }

  void erase(string k) {
    int idx = find(k);
    if(idx == -1) return;
    if(pre[idx] != -1) nxt[pre[idx]] = nxt[idx];
    if(nxt[idx] != -1) pre[nxt[idx]] = pre[idx];
    int h = myHash(k);
    if(head[h] == idx) head[h] = nxt[idx];
  }

  int main(int argc, char **argv){
    fill(head, head+M, -1);
    fill(pre, pre+MX, -1);
    fill(nxt, nxt+MX, -1);

    cout << "start!\n";
    insert("orange", 724);           // ("orange", 724)
    insert("melon", 20);             // ("orange", 724), ("melon", 20)
    assert(val[find("melon")] == 20);
    insert("banana", 52);            // ("orange", 724), ("melon", 20), ("banana", 52)
    insert("cherry", 27);            // ("orange", 724), ("melon", 20), ("banana", 52), ("cherry",
        27)
    insert("orange", 100);           // ("orange", 100), ("melon", 20), ("banana", 52), ("cherry",
        27)
    assert(val[find("banana")] == 52);
    assert(val[find("orange")] == 100);
    erase("wrong_fruit");            // ("orange", 100), ("melon", 20), ("banana", 52), ("cherry",
        27)
    erase("orange");                 // ("melon", 20), ("banana", 52), ("cherry", 27)
    assert(find("orange") == -1);
    erase("orange");                 // ("melon", 20), ("banana", 52), ("cherry", 27)
    insert("orange", 15);            // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
        15)
    assert(val[find("orange")] == 15);
    insert("apple", 36);             // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
        15), ("apple", 36)
    insert("lemon", 6);              // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
        15), ("apple", 36), ("lemon", 6)
    insert("orange", 701);           // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
        701), ("apple", 36), ("lemon", 6)
    assert(val[find("cherry")] == 27);
    erase("xxxxxxx");
    assert(find("xxxxxxx") == -1);
    assert(val[find("apple")] == 36);
    assert(val[find("melon")] == 20);
    assert(val[find("banana")] == 52);
    assert(val[find("cherry")] == 27);
    assert(val[find("orange")] == 701);
    assert(val[find("lemon")] == 6);
    cout << "good!\n";
  }
```

- C++ Skeleton Code (Pure) (Open addressing)

```
  // Hash using open addressing : https://blog.encrypted.gg/1009
  #include <bits/stdc++.h>
  using namespace std;

  const int M = 1000003;
  const int a = 1000;
  const int EMPTY = -1;
  const int OCCUPY = 0;
  const int DUMMY = 1;

  int status[M]; // EMPTY, OCCUPY, DUMMY
  string key[M];
  int val[M];
```

```
14
15      int myHash(string& s){
16        int h=0;
17        for(auto x : s)
18        h = (h*a + x) % M;
19        return h;
20      }
21
22      int find(string k) {
23        int idx = myHash(k);
24        while(status[idx] != EMPTY) {
25          if(status[idx] == OCCUPY && key[idx] == k) return idx;
26          idx = (idx+1) % M;
27        }
28        return -1;
29      }
30
31      void insert(string k, int v) {
32        int idx = find(k);
33        if(idx != -1) {
34          val[idx] = v;
35          return ;
36        }
37        idx = myHash(k);
38        while(status[idx] == OCCUPY)
39        idx = (idx+1) % M;
40
41        key[idx] = k;
42        val[idx] = v;
43        status[idx] = OCCUPY;
44      }
45
46      void erase(string k) {
47        int idx = find(k);
48        if(idx != -1) status[idx] = DUMMY;
49      }
50
51      int main(int argc, char **argv){
52        fill(status, status+M, -1);
53
54        cout << "start!\n";
55        insert("orange", 724);           // ("orange", 724)
56        insert("melon", 20);             // ("orange", 724), ("melon", 20)
57        assert(val[find("melon")] == 20);
58        insert("banana", 52);            // ("orange", 724), ("melon", 20), ("banana", 52)
59        insert("cherry", 27);            // ("orange", 724), ("melon", 20), ("banana", 52), ("cherry",
              27)
60        insert("orange", 100);           // ("orange", 100), ("melon", 20), ("banana", 52), ("cherry",
              27)
61        assert(val[find("banana")] == 52);
62        assert(val[find("orange")] == 100);
63        erase("wrong_fruit");            // ("orange", 100), ("melon", 20), ("banana", 52), ("cherry",
              27)
64        erase("orange");                 // ("melon", 20), ("banana", 52), ("cherry", 27)
65        assert(find("orange") == -1);
66        erase("orange");                 // ("melon", 20), ("banana", 52), ("cherry", 27)
67        insert("orange", 15);            // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
              15)
68        assert(val[find("orange")] == 15);
69        insert("apple", 36);             // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
              15), ("apple", 36)
70        insert("lemon", 6);              // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
              15), ("apple", 36), ("lemon", 6)
71        insert("orange", 701);           // ("melon", 20), ("banana", 52), ("cherry", 27), ("orange",
              701), ("apple", 36), ("lemon", 6)
72        assert(val[find("cherry")] == 27);
73        erase("xxxxxxx");
74        assert(find("xxxxxxx") == -1);
75        assert(val[find("apple")] == 36);
76        assert(val[find("melon")] == 20);
```

```
  assert(val[find("banana")] == 52);
  assert(val[find("cherry")] == 27);
  assert(val[find("orange")] == 701);
  assert(val[find("lemon")] == 6);
  cout << "good!\n";
}
```

- C++ Skeleton Code (w/ STL)

```
// Hash using stl : https://blog.encrypted.gg/1009
#include <bits/stdc++.h>
using namespace std;

unordered_set<int> s;

int main(int argc, char **argv){
  cout << "start!\n";
  s.insert(-10); s.insert(100); s.insert(15);  // {-10, 100, 15}
  s.insert(-10);                               // {-10, 100, 15}
  cout << s.erase(100) << '\n';                // 1
  cout << s.erase(20) << '\n';                 // 0
  if(s.find(15) != s.end()) cout << "15 in s\n"; // 15 in s
  else cout << "15 not in s\n";
  cout << s.size() << '\n';                    // 2
  cout << s.count(50) << '\n';                 // 0
  for(auto e : s) cout << e << ' ';            // 15 -10
  cout << '\n';
  cout << "good!\n";
}
```

1.11 Tree

트리(Tree)는 계층적 관계를 표현하는 비선형 자료구조로, 노드와 노드를 연결하는 엣지로 구성된다. 이진 트리, AVL 트리, Red Black 트리 등 다양한 형태가 있으며, 계층적 데이터를 관리하는데 적합하다.

- 트리: 무방향이면서 사이클이 없는 연결 그래프(undirected acyclic connected graph)
  - V개의 정점을 가지고 $V-1$개의 간선을 가지는 연결 그래프 연결 그래프이면서 임의의 간선을 제거하면 연결 그래프가 아니게 되는 그래프
- 이진 트리(Binary tree) : 각 노드의 자식이 최대 2개 이하인 트리 (레벨/전위/중위/후위 순회를 할 수 있다)
  - 전위/중위/후위 순회를 하는 예제 (C++)

```
#include <bits/stdc++.h>
using namespace std;

struct Node{
  char left;
  char right;
};

int N;
map<char, Node> tree;

// 전위순회
void preOrder(char node){
  if(node=='.') return;
  printf("%c", node);
  preOrder(tree[node].left);
  preOrder(tree[node].right);
}

// 중위순회
void inOrder(char node){
  if(node=='.') return;
```

```
23          inOrder(tree[node].left);
24          printf("%c", node);
25          inOrder(tree[node].right);
26        }
27
28        // 후위순회
29        void postOrder(char node){
30          if(node=='.') return;
31          postOrder(tree[node].left);
32          postOrder(tree[node].right);
33          printf("%c", node);
34        }
35
36        int main(int argc, char **argv){
37          scanf("%d", &N);
38
39          for(int i=0;i<N;i++){
40            char node, left, right;
41            cin >> node >> left >> right;
42            tree[node].left = left;
43            tree[node].right = right;
44          }
45
46          preOrder('A'); printf("\n");
47          inOrder('A'); printf("\n");
48          postOrder('A'); printf("\n");
49        }
```

- 자가 균형 이진 검색 트리 : 트리가 편향되는 걸 방지하기 위해 자체적으로 루트를 변경하여 균형을 맞추는 트리 (AVL, Red Black 트리 등)

1.12 Binary search tree

이진 검색 트리(binary search tree)는 왼쪽 서브트리의 모든 값은 부모의 값보다 작고 오른쪽 서브트리의 모든 값은 부모의 값보다 큰 이진 트리(binary tree)를 말한다. 이진 검색 트리를 활용하면 대부분의 연산을 $O(\log n)$에 처리할 수 있다. 또한 원소가 크기 순으로 정렬되기 때문에 값의 대소와 관련된 성질이 필요한 경우에 유용하게 사용할 수 있다.

- 시간복잡도: insert, erase, find, update 모두 $O(\log n)$

1.12.1 std::set

STL에서 제공하는 set은 자가 균형 이진 검색 트리(self-balancing binary search tree)의 일종인 Red Black 트리 자료구조로 구현되어 있으며 '키' 만을 사용한다. 이는 요소의 중복을 허용하지 않는 모든 경우에 유용하게 사용된다.

- 시간복잡도: `insert`, `erase`, `find`, `lower_bound`, `next`, `prev` 모두 $O(\log n)$
- C++ Skeleton Code (w/ STL)

```
1   // Set : https://blog.encrypted.gg/1013
2   #include <bits/stdc++.h>
3   using namespace std;
4
5   int main(){
6     set<int> s;
7
8     s.insert(-10); s.insert(100); s.insert(15); // {-10, 15, 100}
9     s.insert(-10);                              // {-10, 15, 100}
10    cout << s.erase(100) << '\n';               // {-10, 15}, 1
11    cout << s.erase(20) << '\n';                // {-10, 15}, 0
12    if(s.find(15) != s.end()) cout << "15 in s\n";
13    else cout << "15 not in s\n";
14    cout << s.size() << '\n';                   // 2
15    cout << s.count(50) << '\n';                // 0
16    for(auto e : s) cout << e << ' ';
```

```
    cout << '\n';
    s.insert(-40);                          // {-40, -10, 15}

    set<int>::iterator it1 = s.begin();     // {-40(<-it1), -10, 15}
    it1++;                                  // {-40, -10(<-it1), 15}

    auto it2 = prev(it1);                   // {-40(<-it2), -10, 15}
    it2 = next(it1);                        // {-40, -10, 15(<-it2)}
    advance(it2, -2);                       // {-40(<-it2), -10, 15}

    auto it3 = s.lower_bound(-20);          // {-40, -10(<-it3), 15}

    auto it4 = s.find(15);                  // {-40, -10, 15(<-it4)}

    cout << *it1 << '\n';                   // -10
    cout << *it2 << '\n';                   // -40
    cout << *it3 << '\n';                   // -10
    cout << *it4 << '\n';                   // 15
}
```

1.12.2 std::map

STL에서 제공하는 map은 자가 균형 이진 검색 트리(self-balancing binary search tree)의 일종인 Red Black 트리 자료구조로 구현되어 있으며 '키-값' 쌍을 사용한다. 각 키는 유일하며 이를 통해 값을 조회할 수 있다. 데이터의 효율적인 저장과 검색에 쓰인다.

- 시간복잡도: `insert`, `erase`, `find`, `lower_bound`, `next`, `prev` 모두 $O(\log n)$
- C++ Skeleton Code (w/ STL)

```
// Map : https://blog.encrypted.gg/1013
#include <bits/stdc++.h>
using namespace std;

int main(){
  map<string, int> m;

  m["hi"] = 123;
  m["bkd"] = 1000;
  m["gogo"] = 165;                          // ("bkd", 1000), ("gogo", 165), ("hi", 123)
  cout << m.size() << '\n';                 // 3
  m["hi"] = -7;                             // ("bkd", 1000), ("gogo", 165), ("hi", -7)

  if(m.find("hi") != m.end()) cout << "hi in m\n";
  else cout << "hi not in m\n";

  m.erase("bkd");                           // ("gogo", 165), ("hi", 123)

  for(auto e : m)
  cout << e.first << ' ' << e.second << '\n';

  auto it1 = m.find("gogo");
  cout << it1->first << ' ' << it1->second << '\n'; // gogo 165
}
```

1.13 Segment tree

구간 트리(segment tree)는 배열 같은 선형 자료구조에 대해 구간 합이나 구간 최댓값 · 최솟값 등의 쿼리를 빠르게 처리하기 위해 트리 형태로 분할 · 구성하는 자료구조이다. 배열을 여러 구간으로 나누어 각 구간의 대표 정보를 저장하며, 쿼리가 들어오면 관련된 구간들만 빠르게 확인하여 결과를 합산하거나 갱신한다. 이를 통해 원소를 일일이 확인하지 않고도 구간 정보를 효율적으로 처리할 수 있으며, 쿼리와 업데이트 모두 일반적으로 $O(\log n)$에 처리할 수 있다.

- C++ Skeleton Code (Pure)

```c
// JUNGOL 1726 구간의 최대값1
// jungol.co.kr/problem/1726
// 세그먼트 트리 : 업데이트 및 쿼리
#include <cstdio>

const int LM = 50005;
const int TLM = 1 << 17; // 131072

int N, Q;
int tree[TLM];

int max(int a, int b) { return a > b ? a : b; }

// 세그먼트 트리 업데이트 함수
void update(int node, int s, int e, int tg, int val) {
  if (s >= e) { // 리프 노드에 도달했을 때 값 업데이트
    tree[node] = val;
    return;
  }
  int m = (s + e) / 2, lch = node * 2, rch = lch + 1;
  if (tg <= m) update(lch, s, m, tg, val); // 왼쪽 자식 노드 업데이트
  else update(rch, m + 1, e, tg, val);  // 오른쪽 자식 노드 업데이트
  tree[node] = max(tree[lch], tree[rch]); // 부모 노드는 자식 노드 중 최댓값을 저장
}

// 구간 내 최댓값을 찾는 쿼리 함수
int query(int node, int s, int e, int qs, int qe) {
  if (e < qs || qe < s) return -1;       // 쿼리 범위 밖이면 무효값 반환
  if (qs <= s && e <= qe) return tree[node]; // 현재 구간이 완전히 쿼리 범위에 포함되면 반환
  int m = (s + e) / 2, lch = node * 2, rch = lch + 1;
  int leftMax = query(lch, s, m, qs, qe); // 왼쪽 서브트리 탐색
  int rightMax = query(rch, m + 1, e, qs, qe); // 오른쪽 서브트리 탐색
  return max(leftMax, rightMax);         // 두 값 중 최댓값 반환
}

int main() {
  scanf("%d %d", &N, &Q);
  int i, s, e, val;

  // 초기 데이터 입력 및 트리 업데이트
  for (i = 1; i <= N; ++i) {
    scanf("%d", &val);
    update(1, 1, N, i, val);
  }

  // 쿼리 처리
  for (i = 0; i < Q; ++i) {
    scanf("%d %d", &s, &e);
    printf("%d\n", query(1, 1, N, s, e));
  }
}
```

- C++ Skeleton Code (Pure, 구조체 버전)

```c
// JUNGOL 1726 구간의 최대값1
// jungol.co.kr/problem/1726
#include <bits/stdc++.h>
#define FASTIO cin.tie(0); cout.tie(0); ios_base::sync_with_stdio(0);
using namespace std;

int N, Q;

struct SegTree{
  int n;               // 배열의 길이
  vector<int> tree;    // 각 구간의 최소치

  SegTree(const vector<int>& arr) {
    n = arr.size();
    tree.resize(n*4);
```

```
16          init(arr, 0, n-1, 1);
17        }
18        // node가 arr[l ... r] 배열을 표현할 때 node를 루트로 하는 서브트리를 초기화하고 이 구간의 최소치를
              반환한다
19        int init(const vector<int>& arr, int s, int e, int node){
20          if(s==e) return tree[node] = arr[s];
21
22          int m = (s+e)/2;
23          int smin = init(arr, s, m, node*2);
24          int emin = init(arr, m+1, e, node*2+1);
25          return tree[node] = max(smin, emin);
26        }
27
28        int query(int s, int e, int node, int qs, int qe) {
29          if(e < qs || qe < s) return INT_MIN;
30          if(qs >= s && qe <= e) return tree[node];
31
32          int m = (qs+qe)/2;
33          return max( query(s, e, node*2, qs, m),
34          query(s, e, node*2+1, m+1, qe) );
35        }
36
37        int query(int s, int e) { return query(s, e, 1, 0, n-1); }
38
39        int update(int idx, int nval, int node, int qs, int qe) {
40          // idx가 노드가 표현하는 구간과 상관없는 경우엔 무시한다
41          if(idx < qs || qe < idx) return tree[node];
42          if(qs == qe) return tree[node] = nval;
43
44          int m = (qs+qe)/2;
45          return tree[node] = max(update(idx, nval, node*2, qs, m),
46          update(idx, nval, node*2+1, m+1, qe) );
47        }
48
49        int update(int idx, int nval) { return update(idx, nval, 1, 0, n-1); }
50      };
51
52      int main(int argc, char **argv){
53        FASTIO;
54        cin >> N >> Q;
55
56        vector<int> arr(N);
57        for(int i = 0; i < N; i++) {
58          cin >> arr[i];
59        }
60
61        SegTree seg(arr);
62
63        while (Q--) {
64          int s, e; cin >> s >> e;
65          int ans = seg.query(s - 1, e - 1);
66          cout << ans << '\n';
67        }
68      }
```

1.13.1 Fenwick tree

펜윅 트리(Fenwick tree), 또는 바이너리 인덱스 트리(Binary Indexed Tree)는 선형적인 배열 구조에서 특정 위치의 값 갱신과 구간의 누적 합(prefix sum) 등의 쿼리를 효율적으로 처리하기 위한 자료구조이다. 배열을 인덱스의 비트 단위로 적절히 나누어, 각 구간에 대한 부분적인 합 정보를 트리 형태로 압축하여 저장한다. 쿼리가 발생하면 이 비트 연산을 이용해 필요한 구간들만 효율적으로 탐색하고 결과를 빠르게 누적하거나 갱신한다. 이를 통해 단순 배열 탐색 없이도 빠르게 구간 정보를 처리할 수 있으며, 각 원소 업데이트와 누적합 쿼리를 모두 일반적으로 $O(\log N)$ 시간에 처리할 수 있다.

- 세그먼트 트리(Segment tree)보다 구현이 훨씬 간단하고 메모리 사용량도 적으며 상수 시간이 작아 더 빠르게 동작하는 경우가 많다.

- 하지만 세그먼트 트리가 다양한 연산(구간 합, 최대/최소, GCD 등)에 쉽게 확장 가능한 반면, 펜윅 트리는 주로 구간 합과 빈도수 관리 등 비교적 단순한 연산에 특화되어 있다. 펜윅 트리는 기본적으로 누적합 구조(prefix sum)를 기반으로 하므로 임의의 복잡한 연산을 처리하려면 추가적인 변형이나 테크닉이 필요하다.

- C++ Skeleton Code (Pure, 구조체 버전)

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Fenwick {
  vector<int> tree;
  Fenwick(int n) : tree(n+1) {}

  int sum(int pos){
    ++pos;
    int ret=0;
    while(pos>0){
      ret += tree[pos];
      pos &= (pos-1);
    }
    return ret;
  }

  void add(int pos, int val){
    ++pos;
    while(pos < tree.size()){
      tree[pos] += val;
      pos += (pos & -pos);
    }
  }

  int range_sum(int s, int e) {
    return sum(e) - sum(s-1);
  }
};

int main() {
  int N = 8;                                  // 원소 8개
  vector<int> A = {0, 3, 2, 4, 1, 6, 5, 7, 2}; // 1-based dummy 0 포함
  // idx:    1  2  3  4  5  6  7  8

  Fenwick ft(N);

  // build
  for(int i=1; i<=N; i++) ft.add(i, A[i]);

  printf("%d\n", ft.range_sum(2,5)); // 2~5 합 = 2+4+1+6 = 13
  ft.add(4,3);                       // A[4] += 3 (4→7)
  printf("%d\n", ft.range_sum(2,5)); // 2~5 합 = 16
}
```

1.14 Graph

그래프(Graph)는 노드들과 이를 연결하는 엣지들로 구성된 자료구조이다. 방향성, 가중치 유무에 따라 다양한 종류가 있으며, 네트워크 구조, 경로 찾기 등에 사용된다.

- 차수(degree) : 각 노드에 대해 엣지로 연결된 이웃한 노드의 개수
- 무방향 그래프(undirected graph) : 엣지의 방향이 없는 그래프 (degree 존재)
- 방향 그래프(directed graph): 엣지의 방향이 있는 그래프 (indegree, outdegree 존재)
- 사이클 : 임의의 한 점에서 출발해 자기 자신으로 돌아올 수 있는 경로
- 순환 그래프(cyclic graph) : 사이클이 하나라도 있는 그래프

- 비순환 그래프(acyclic graph) : 사이클이 하나도 없는 그래프
- 완전 그래프(complete graph) : 모든 서로 다른 두 노드 쌍이 엣지로 연결된 그래프
- 연결 그래프(connected graph) : 임의의 두 노드 사이에 경로가 항상 존재하는 그래프
- 루프 : 한 노드에서 시작해 같은 노드로 들어오는 엣지
- C++ Skeleton Code (Pure) (인접 행렬)

```cpp
#include <bits/stdc++.h>
using namespace std;

int adj[10][10] = {};

int main(int argc, char **argv){
  int v,e;
  scanf("%d %d", &v, &e);
  for(int i=0; i<e; i++){
    int u,v;
    scanf("%d %d", &u, &v);
    adj[u][v] = 1;
    adj[v][u] = 1;  // 무방향 그래프 (방향 그래프이면 12번 라인만 사용)
  }
}
```

- C++ Skeleton Code (Pure) (인접 리스트)

```cpp
#include <bits/stdc++.h>
using namespace std;

int edge[10][2];
int deg[10];   // 각 정점의 outdegree
int* adj[10];
int idx[10];   // adj[i]에서 새로운 정점을 추가할 때의 위치

int main(int argc, char **argv){
  int v,e;
  scanf("%d %d", &v, &e);
  for(int i=0; i<e; i++){
    scanf("%d %d", &edge[i][0], &edge[i][1]);
    deg[edge[i][0]]++;
    deg[edge[i][1]]++; // 무방향 그래프 (방향 그래프인 경우 14번 라인만 사용)
  }

  for(int i=1; i<=v; i++)
  adj[i] = new int[deg[i]];

  for(int i=0; i<e; i++){
    int u = edge[i][0], v = edge[i][1];
    adj[u][idx[u]] = v;
    idx[u]++;
  }
}
```

- C++ Skeleton Code (w/ STL) (인접 리스트)

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> adj[10];

int main(int argc, char **argv){
  int v,e;
  scanf("%d %d", &v, &e);
  for(int i=0; i<e; i++){
    int u,v;
    scanf("%d %d", &u, &v);
    adj[u].push_back(v);
```

```
      adj[v].push_back(u);  // 무방향 그래프 (방향 그래프이면 12번 라인만 사용)
    }
  }
```

- 공간복잡도: 인접 행렬 $O(V^2)$, 인접 리스트 $O(V+E)$
- 인접 행렬 구현은 두 점의 연결여부를 자주 확인하거나 E가 V^2와 가까울 때 사용하면 효율적
- 인접 리스트 구현은 특정 정점에 연결된 모든 정점을 자주 확인하거나 E가 V^2보다 훨씬 작을 때 효율적이다 (일반적으로 인접 리스트가 필요한 상황이 PS에서 자주 나옴)

1.15 Union-find

유니온 파인드(union find) 또는 서로소 집합(disjoint set union, DSU)는 여러 개의 원소를 효율적으로 그룹화하고 관리하는 자료구조로, "find" 연산을 통해 특정 원소가 속한 집합의 대표를 찾고, "Union" 연산을 통해 두 집합을 합칠 수 있다. 경로 압축(path compression)과 랭크 기반 합치기(union by rank) 같은 최적화 기법을 적용하면 평균적으로 매우 빠르게 동작하며, 그래프의 연결 요소 찾기, 최소 스패닝 트리(MST) 등 다양한 문제에서 활용된다.

- 시간복잡도: 최적화 적용 시 $O(\alpha(n))$
  - $\alpha(n)$: 아커만 함수, $\alpha(10^{80}) = 5$ 수준이므로 사실 상 상수 시간복잡도라고 보면 된다
- C++ Skeleton Code (Pure) #1

```cpp
// 종교 jungol.co.kr/problem/1863
#include <bits/stdc++.h>
using namespace std;

constexpr int LM = 50005;
int N, M, ret;
int p[LM], rk[LM];

int find(int x) {
  if (p[x] == x) return x;
  return p[x] = find(p[x]); // 최적화1 : 경로 압축 적용
}

void swap(int& a, int& b) { int c=a; a=b; b=c;}

void Union( int a, int b ) {
  a = find( a ), b = find( b );
  if ( a == b ) return;
  ret--;

  // 최적화2 : 랭크 기반 합치기
  if ( rk[a] < rk[b] ) swap( a, b );
  p[b] = a;
  if ( rk[a] == rk[b] ) rk[a]++;
}

int main() {
  scanf( "%d %d", &N, &M );
  ret = N;

  for(int i=0; i<N; i++){
    p[i] = i;
    rk[i] = 0; // 랭크 초기화
  }

  for ( int i = 0; i < M; i++ ) {
    int a, b;
    scanf( "%d %d", &a, &b );
    Union( a, b );
  }
  printf( "%d", ret );
}
```

- C++ Skeleton Code (Pure) #2

```cpp
// 종교 jungol.co.kr/problem/1863
#include <bits/stdc++.h>
using namespace std;

constexpr int LM = 50005;
int N, M, ret;
vector<int> p(LM, -1);

int find(int x) {
  if (p[x] < 0) return x;
  return p[x] = find(p[x]); // 최적화1 : 경로 압축 적용
}

void swap(int& a, int& b) { int c=a; a=b; b=c;}

void Union( int a, int b ) {
  a = find( a ), b = find( b );
  if ( a == b ) return;
  ret--;

  // 최적화2 : 랭크 기반 합치기, rank, parent를 하나의 변수에서 처리하는 버전
  if ( p[a] < p[b] ) swap( a, b );
  if ( p[a] == p[b] ) p[a]--;
  p[b] = a;
}

int main() {
  scanf( "%d %d", &N, &M );
  ret = N;

  for ( int i = 0; i < M; i++ ) {
    int a, b;
    scanf( "%d %d", &a, &b );
    Union( a, b );
  }
  printf( "%d", ret );
}
```

1.16 Trie

Trie(트라이)는 문자열을 효율적으로 저장하고 검색하는 트리형 자료구조로, 각 노드는 문자열의 접두사를 공유하며 계층적으로 구성된다. "삽입(insert)" 연산을 통해 문자를 추가하고, "탐색(search)" 연산을 통해 특정 문자열이 존재하는지 빠르게 확인할 수 있다. 노드에 값을 저장하거나 마킹하는 방식으로 단어의 끝을 나타내며, 해시맵 기반 구현 또는 배열을 활용한 최적화 기법을 적용하면 $O(|L|)$의 시간 복잡도로 동작한다. Trie는 자동완성, 사전(dictionary), 문자열 검색 등 다양한 문제에서 활용된다.

- $|L|$: 문자열 L의 길이
- 시간복잡도: insert, erase, find 모두 $O(|L|)$
- 문자열을 그냥 배열에 저장하는 것보다 최대 '4 x 글자의 종류' 배만큼 메모리를 더 사용한다 (e.g., 각 단어가 알파벳 대문자로만 구성되어 있을 경우 104배)
- 이론적인 시간복잡도와는 별개로 실제로는 트라이가 해시, 이진검색트리에 비해 훨씬 느리며 공간복잡도도 훨씬 크다. 일반적인 문자열 문제는 해시, 이진검색트리를 사용하면 되지만 자동완성 같은 특수한 활용에서는 트라이가 필요하다

- C++ Skeleton Code (Pure)

```cpp
// 문자열 집합 boj.kr/14425
#include <bits/stdc++.h>
using namespace std;

const int LM = 500 * 10000 + 5; // 최대 등장 가능한 글자의 수 (길이 500인 문자열 10000개)
```

```
const int ROOT = 1;

int N, M;
int unused = 2;
bool chk[LM];   // 해당 정점이 문자열의 끝인지 여부 저장
int nxt[LM][26]; // 각 정점에서 자식의 정점 번호
// 배열에 문자열 저장 : char 1칸 (1바이트)
//Trie에 문자열 저장 : int 26칸 (4x26바이트)

int c2i(char c) {  return c-'a'; }

// Trie
void insert( string& s ) {
  int cur = ROOT;
  for ( auto c : s ) {
    int i = c2i( c );
    if ( nxt[cur][i] == -1 )
    nxt[cur][i] = unused++;
    cur = nxt[cur][i];
  }
  chk[cur] = true;
}

bool find( string& s ) {
  int cur = ROOT;
  for ( auto c : s ) {
    int i = c2i( c );
    if ( nxt[cur][i] == -1 ) return false;
    cur = nxt[cur][i];
  }
  return chk[cur];
}

void erase( string& s ) {
  int cur = ROOT;
  for ( auto c : s ) {
    int i = c2i( c );
    if ( nxt[cur][i] == -1 ) return;
    cur = nxt[cur][i];
  }
  chk[cur] = false;
}

int main( int argc, char** argv ) {
  for ( int i = 0; i < LM; i++ ) {
    fill( nxt[i], nxt[i] + 26, -1 );
  }

  scanf( "%d%d", &N, &M );

  while ( N-- ) {
    string s;
    cin >> s;
    insert( s );
  }

  int ans = 0;
  while ( M-- ) {
    string s;
    cin >> s;
    ans += find( s );
  }
  printf( "%d\n", ans );
}
```

- C++ Skeleton Code (Pure, 구조체 버전)

```
// 문자열 집합 boj.kr/14425
#include <bits/stdc++.h>
using namespace std;
```

```
4
5   const int LM = 500*10000+5;
6   int N,M;
7
8   int c2i(char c) { return c-'a'; }
9
10  struct TrieNode {
11    TrieNode* child[26];
12    bool terminal;
13
14    TrieNode() :terminal(false){
15      memset(child, 0, sizeof(child));
16    }
17    ~TrieNode(){
18      for(int i=0; i<26; i++){
19        if(child[i]) delete child[i];
20      }
21    }
22
23    void insert(const char* key){
24      if(*key==0) terminal = true;
25      else {
26        int next = c2i(*key);
27        if(child[next] == NULL) child[next] = new TrieNode();
28        child[next]->insert(key+1);
29      }
30    }
31
32    TrieNode* find(const char* key){
33      if(*key==0) return this;
34      int next = c2i(*key);
35      if(child[next] == NULL) return NULL;
36      return child[next]->find(key+1);
37    }
38  };
39
40
41  int main(int argc, char **argv){
42    //freopen("s_in_1345.txt", "r", stdin);
43
44    scanf("%d%d",&N,&M);
45
46    TrieNode trie;
47
48    while(N--){
49      string s; cin >> s;
50      trie.insert(s.c_str());
51    }
52
53    int ans = 0;
54
55    while(M--) {
56      string s; cin >> s;
57      TrieNode* t = trie.find(s.c_str());
58      if(t != nullptr && t->terminal) ans++;
59    }
60
61    printf("%d\n", ans);
62  }
```

2 Algorithm

2.1 Math

2.1.1 Prime number

소수(prime number)는 1과 자기 자신으로만 나누어지는 수를 말한다. 즉, 임의의 수를 약분했을 때 약수가 2개이면 해당 수를 소수라고 한다.

- 현재 숫자 n이 1 이외에 다른 수로 나눠지는지 확인함으로써 소수 판별
- 시간복잡도: $O(\sqrt{n})$

```
bool isPrime(int n) {
  if(n==1) return 0;
  for(int i=2; i*i<=n; i++){ // search until sqrt(n)
    if(n % i == 0) return 0;
  }
  return 1;
}
```

- 에라토스테네스의 체(Sieve of Eratosthenes)를 사용하여 소수가 아닌 값들을 마킹함으로써 소수 판별 (시간복잡도: $O(n \log(\log n))$)

```
#include<bits/stdc++.h>

constexpr int N = 1005;
bool is_prime[N];

// Pure
void sieve() {
  memset(is_prime, 1, sizeof(is_prime));
  is_prime[1] = false;
  for(int i=2; i*i<=N; i++){
    if(is_prime[i] == false) continue;
    for(int j=i*i; j<=N; j+=i) {
      is_prime[j] = false;
    }
  }
}

int main() {
  sieve();

  for(int i=2; i<=N; i++){
    if(is_prime[i]) {  // check if 'i' is prime number
      printf("%d ", i);
    }
  }puts("");
}
```

```
#include<bits/stdc++.h>
using namespace std;

int N = 1005;
vector<int> primes;

// STL
void sieve() {
  vector<bool> state(N+1, true);
  state[1] = false;
  for(int i=2; i*i<=N; i++){
    if(!state[i]) continue;
    for(int j=i*i; j<=N; j+=i)
    state[j] = false;
  }
  for(int i=2; i<=N; i++) {
    if(state[i]) primes.push_back(i);
```

```
    }
  }

  int main() {
    sieve();

    for(int i=2; i<=N; i++){
      // check if 'i' is prime number
      if(find(primes.begin(), primes.end(), i) != primes.end()) {
        printf("%d ", i);
      }
    } puts("");
  }
```

2.1.2 Prime factorization

소인수분해(prime factorization)은 정수를 소수의 곱으로 나타내는 것을 의미한다. 모든 자연수는 소인수분해하는 방법이 딱 한가지만 존재하므로 유일한 소인수분해 값을 얻을 수 있다.

- C++ Skeleton Code (Pure) (시간복잡도: $O(\sqrt{n})$)

```
#include <bits/stdc++.h>
using namespace std;

// O(sqrt(n))
void primeFactorization(int n) {
  for(int i=2; i*i<=n; i++){
    while(n%i == 0) {
      printf("%d ", i);
      n /= i;
    }
  }
  if(n != 1) printf("%d ", n);
  puts("");
}

int main(int argc, char **argv){
  int N;
  scanf("%d",&N); // 1100

  primeFactorization(N); // 2 2 5 5 11
}
```

- 에라토스테네스의 체를 활용한 빠른 소인수분해 (시간복잡도: $O(\log n)$)

```
#include <bits/stdc++.h>
using namespace std;

const int LM = 1000005;

int minFactor[LM]; // minFactor[i] :i의 가장 작은 소인수(i가 소수인 경우는 자기 자신)

// 시간복잡도: O(nloglogn)
void eratosthenes2(int n) {
  //초기화
  minFactor[0] = minFactor[1] = -1;
  for ( int i = 2; i <= n; i++ ) {
    minFactor[i] = i;
  }

  // 에라토스테네스의 체
  for ( int i = 2; i*i <= n; i++ )
  if ( minFactor[i] == i )
  for ( int j = i * i; j <= n; j += i )
  if ( minFactor[j] == j )     // 아직 약수를 본 적 없는 숫자인 경우 i를 써둔다
  minFactor[j] = i;
}

```

```
  // 시간복잡도: O(logn)
  vector<int> factor( int n ) {
    vector<int> ret;
    // n이 1이 될 때까지 가장 작은 소인수로 나누기를 반복한다
    while ( n > 1 ) {
      ret.push_back( minFactor[n] );
      n /= minFactor[n];
    }
    return ret;
  }

  int main( int argc, char **argv ) {
    int N;
    scanf("%d", &N); // 1100

    eratosthenes2(N);
    vector<int> ret = factor(N);

    for(auto& e : ret) {
      printf("%d ", e); // 2 2 5 5 11
    }
    puts("");
  }
```

2.1.3 Divisor

약수(divisor)는 어떤 수를 나누어떨어지게 하는 수를 말한다. 예를 들어 24의 약수는 $[1, 2, 3, 4, 6, 8, 12, 24]$ 이다.

- C++ Skeleton Code (w/ STL) (시간복잡도 $O(\sqrt{n})$)

```
#include <bits/stdc++.h>
using namespace std;

// O(sqrt(n))
vector<int> divisor(int n){
  vector<int> div;
  for(int i=1; i*i <= n; i++){
    if(n % i == 0) div.push_back(i);
  }
  for(int j=(int)div.size()-1; j>=0; j--) {
    if(div[j]*div[j] == n) continue;
    div.push_back(n/div[j]);
  }
  return div;
}

int main(int argc, char **argv){
  int N;
  scanf("%d",&N); // 24

  vector<int> divs = divisor(N);

  for(auto x : divs) printf("%d ", x); // 1 2 3 4 6 8 12 24
  puts("");
}
```

2.1.4 GCD

최대공약수(greatest common divisor)는 두 자연수의 공통된 약수 중 가장 큰 약수를 말한다. 유클리드 호제법을 사용하면 GCD를 빠르게 구할 수 있다.

- 유클리드 호제법: 두 수 a, b에 대하여 a를 b로 나눈 나머지를 r이라고 하면 GCD(a, b)=GCD(b, r)이 성립한다
- C++ Skeleton Code (Pure) (시간복잡도: $O(\log(\min(a, b)))$)

```cpp
#include <bits/stdc++.h>
using namespace std;

int gcd(int a, int b){
  if(a==0) return b;
  return gcd(b%a, a);
}

int main(int argc, char **argv){
  int A,B;
  scanf("%d %d",&A, &B); // 32, 6

  printf("%d\n", gcd(A, B)); // GCD(32,6) = 2
}
```

2.1.5 LCM

최소공배수(least common multiple)는 두 자연수의 공통된 배수 중 가장 작은 배수를 말한다.

- $\mathrm{LCM}(a,b) = (a \times b) / \mathrm{GCD}(a,b)$가 성립한다 (원래 식은 $a \times b = \mathrm{GCD}(a,b) \cdot \mathrm{LCM}(a,b)$)
- C++ Skeleton Code (Pure) (시간복잡도: $O(\log(\min(a,b)))$)

```cpp
#include <bits/stdc++.h>
using namespace std;

int gcd(int a, int b){
  if(a==0) return b;
  return gcd(b%a, a);
}

int lcm(int a, int b){
  return a / gcd(a,b)*b;
}

int main(int argc, char **argv){
  int A,B;
  scanf("%d %d",&A, &B); // 32, 6

  printf("%d\n", lcm(A, B)); // LCM(32,6) = 96
}
```

2.1.6 Permutation

순열(permutation)은 순서가 부여된 임의의 집합을 다른 순서로 뒤섞는 연산이다.

- 시간복잡도: $O(n! \times n)$
- C++ Skeleton Code (w/ STL)

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
constexpr int LM=10005;
int num[LM] = {0};
int main(int argc, char **argv){
  num[0]=1;
  num[1]=2;
  num[2]=3;
  do{
    for(int i=0;i<3;i++)
    printf("%d ", num[i]);
    printf("\n");
  }while(next_permutation(num, num+3));
  return 0;
}
```

- C++ Skeleton Code (Pure)

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;

int num[3]={0};
int N=3;

int main(int argc, char **argv){
  num[0]=1, num[1]=2, num[2]=3;

  while(1) {
    for(int i=0;i<N;i++)
    printf("%d ", num[i]);
    printf("\n");

    bool last_perm=true;    // 마지막 순열인지 체크

    for(int i=N-1;i>0;i--){ // 입력된 순열의 마지막 원소부터 검사
      if(num[i-1] < num[i]){ // 왼쪽 원소(기준) < 오른쪽 원소
        int idx=i;         // 교환할 원소의 인덱스
        for(int j=N-1; j>=i; j--)
        if(num[i-1] < num[j] && num[j] < num[idx]) // 기준 원소보다 크면서 제일 작은 원소
        idx=j;

        int tmp = num[idx]; // 교환
        num[idx] = num[i-1];
        num[i-1] = tmp;

        sort(num+i, num+N); // 기준 원소 우측 오름차순 정렬

        last_perm=false;
        break;
      }
    }
    if(last_perm) break;
  }
}
```

- 순열 ${}_nP_r$ 경우의 수를 구하는 예제 (C++)

```cpp
#include <bits/stdc++.h>

// nPr = n! / (n-r)!
// 시간복잡도 O(r)
long long permutation(int n, int r) {
  if (r > n) return 0;
  unsigned long long res = 1;
  for (int i = 0; i < r; ++i) {
    res *= (n - i);
  }
  return res;
}

int main() {
  int n = 10, r = 5;
  printf("%lld\n", permutation(n,r)); // 30240
}
```

2.1.7 Combination

조합(combination)은 집합에서 일부 원소를 취해 부분 집합을 만드는 연산이다.

- 시간복잡도: $O(\ {}_nC_k \times n\ ) = O((n!/(k!(n-k)!)) \times n)$
- C++ Skeleton Code (w/ STL)

```cpp
#include <bits/stdc++.h>
using namespace std;

int num[5] = {1,2,3,4,5};

int main(int argc, char **argv) {
  int n = 5;
  int k = 3;

  // Initialize mask with k 1's followed by n-k 0's
  int mask[5];
  memset(mask, 0, sizeof(mask));
  for(int i=0; i<k; i++){
    mask[i] = 1;
  }

  do {
    for (int i = 0; i < n; ++i) {
      if (mask[i])
      printf("%d ", num[i]);
    }
    printf("\n");
  } while (prev_permutation(mask, mask + n));

  // 1 2 3
  // 1 2 4
  // 1 2 5
  // 1 3 4
  // 1 3 5
  // 1 4 5
  // 2 3 4
  // 2 3 5
  // 2 4 5
  // 3 4 5
}
```

- C++ Skeleton Code (Pure) #1

```cpp
#include <cstdio>
using namespace std;

int arr[5] = {1, 2, 3, 4, 5};
int n = 5, k = 3;

int main(int argc, char **argv) {
  int indices[3] = {0,1,2}; // Initialize indices for the first combination

  do {
    for (int i = 0; i < k; ++i)
    printf("%d ", arr[indices[i]]);
    printf("\n");

    int i = k - 1;
    while (i >= 0 && indices[i] == n - k + i)
    --i;

    if (i < 0) break; // All combinations generated

    ++indices[i];
    for (int j = i + 1; j < k; ++j)
    indices[j] = indices[j - 1] + 1;

  } while (1);
}
```

- C++ Skeleton Code (Pure) #2

```cpp
#include <cstdio>
```

```
    using namespace std;

    int arr[5] = {1, 2, 3, 4, 5};
    int sel[5];
    int n = 5, k = 3;

    void comb(int idx, int start) {
      if(idx == k) {
        for(int i = 0; i < k; i++) {
          printf("%d ", sel[i]);
        }
        puts("");

        return;
      }

      for(int i = start; i < n; i++) {
        sel[idx] = arr[i];
        comb(idx + 1, i + 1);
      }
    }

    int main(){
      comb(0, 0);
      return 0;
    }
```

- 조합 ${}_nC_r$ 경우의 수를 구하는 예제 (C++)

```
    #include <bits/stdc++.h>

    using ll = long long;

    const int LM=1005;
    const int MOD=10007;

    ll comb[LM][LM];

    // O(nr)
    void computeCombination() {
      // compute Comb 1 ~ LM in advance
      for(int i=1; i<LM; i++){
        comb[i][0] = comb[i][i] = 1;
        for(int j=1; j<i; j++){
          comb[i][j] = comb[i-1][j] + comb[i-1][j-1]; // DP, nCr = n-1Cr-1 + n-1Cr
        }
      }
    }

    int main(int argc, char **argv){
      int n=10,r=5;
      computeCombination();
      printf("%lld\n", comb[n][r]); // 252
    }
```

2.1.8 Recursion

재귀(recursion)는 함수가 자기 자신을 호출하여 문제를 해결하는 방식이다. 재귀로 문제를 푼다는 것은 귀납적인 방식으로 문제를 해결하겠다는 것과 동일하다. 올바른 재귀 함수는 반드시 특정 입력에 대해서는 자기 자신을 호출하지 않고 종료되어야 하는데 이를 보통 base condition이라고 한다. 또한 모든 입력은 base condition으로 수렴해야 한다. 재귀는 반복문으로 구현했을 때와 비교하여 코드는 간결하지만 메모리와 시간 측면에서는 일반적으로 손해를 본다. 재귀는 분할 정복 알고리즘, 트리 탐색 등에 널리 사용된다.

- 피보나치 수열을 재귀로 구하는 예제 (C++)

```
    #include <cstdio>

```

```
// recursive: O(1.618^n)
int fibonacci(int n) {
  if (n == 0 || n == 1) return 1; // base case
  return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
  int n = 20;
  for (int i = 0; i <= n; i++) {
    printf("%d ", fibonacci(i)); // 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
        4181 6765 10946
  }
  puts("");
}
```

- 하노이 탑을 재귀로 구하는 예제 (C++)

```
#include <bits/stdc++.h>
using namespace std;

void hanoi(int from, int to, int n){
  if(n==1) { printf("%d %d\n",from,to); return; }

  int by = 6-from-to;

  hanoi(from, by, n-1);
  printf("%d %d\n", from,to);
  hanoi(by, to, n-1);
}

void hanoi2(int from, int by, int to, int n){
  if(n==1) { printf("%d %d\n", from, to); return; }

  hanoi2(from, to, by, n-1);              // n-1개의 원반을 보조 기둥(by)로 이동
  printf("%d %d\n", from, to);            // 가장 큰 원반을 목표 기둥(to)로 이동
  hanoi2(by, from, to, n-1);              // n-1개의 원반을 목표 기둥(to)로 이동
}

int main(int argc, char **argv){
  int K; scanf("%d",&K);
  printf("%d\n", (1<<K)-1);

  hanoi(1, 3, K);
  // hanoi2(1, 2, 3, K);
}
```

2.2 Sort

2.2.1 Bubble sort

버블 정렬(bubble sort)는 인접한 두 원소를 비교하여 정렬하는 방식이다. 시간 복잡도가 높아 실제 PS에서는 잘 사용되지 않는다.

- 시간복잡도: 평균/최악 $O(n^2)$, 최선(거의 정렬된 경우) $O(n)$
- 공간복잡도: $O(1)$
- C++ Skeleton Code (Pure)

```
#include <cstdio>
constexpr int LM = 10;

int A[LM] = { 5,2,7,1,3,8,4,6,10,9 };

void BubbleSort(int *arr, int s, int e) {
  for(int i = s; i < e; i++) {
    for(int j = s; j < e - (i - s); j++) {
      if(arr[j] > arr[j + 1]) {
```

```
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
          }
        }
      }
    }

    int main() {
      for(int i=0; i<LM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
      puts("");

      BubbleSort(A, 0, LM - 1);

      for(int i=0; i<LM; i++) printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9 10
      puts("");

      return 0;
    }
```

2.2.2 Insertion sort

삽입 정렬(insertion sort)는 각 원소를 이미 정렬된 배열의 적절한 위치에 삽입하는 방식으로 구현된다. 작은 데이터 집합에 효율적이다.

- 시간복잡도: 평균/최악 $O(n^2)$, 최선(거의 정렬된 경우) $O(n)$
- 공간복잡도: $O(1)$
- C++ Skeleton Code (Pure)

```
#include <cstdio>
constexpr int LM = 10;

int A[LM] = { 5,2,7,1,3,8,4,6,10,9 };


void InsertionSort(int *arr, int s, int e) {
  for(int i = s + 1; i <= e; i++) {
    int key = arr[i];
    int j = i - 1;
    while(j >= s && arr[j] > key) {
      arr[j + 1] = arr[j];
      j--;
    }
    arr[j + 1] = key;
  }
}

int main() {
  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  puts("");

  InsertionSort(A, 0, LM - 1);

  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9 10
  puts("");

  return 0;
}
```

2.2.3 Heap sort

힙(heap) 자료 구조를 사용하여 정렬하는 방식으로 최대 힙(max heap)을 구성한 후 힙의 루트부터 모든 원소를 꺼내서 재구성하면 오름차순으로 정렬된다.

- 시간복잡도: 최악의 경우에도 $O(n \log n)$을 보장하며 추가 메모리 사용이 적은 편 (힙 구성 $O(n)$, n번에 걸쳐서 힙 재정렬 $O(\log n)$ 씩)
- 공간복잡도: $O(1)$ (배열 내에서 힙을 구성하므로 추가 공간이 거의 필요 없음)
- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>

constexpr int NUM = 10;
constexpr int LM = 105;

int A[] = { 5,2,7,1,3,8,4,6,10,9 };

int heap[LM];
int hn = 0;

void swap(int &a, int &b) { int t = a; a = b; b = t; }

void push(int nd) {
  heap[++hn] = nd;

  for (int c = hn; c > 1; c /= 2) {
    if (heap[c] > heap[c / 2]) {
      swap(heap[c], heap[c / 2]);
    }
    else {
      break;
    }
  }
}

void pop() {
  swap(heap[1], heap[hn--]);

  for (int c = 2; c <= hn; c *= 2) {
    if (c < hn && heap[c + 1] > heap[c]) {
      c++;
    }
    if (heap[c] > heap[c / 2]) {
      swap(heap[c], heap[c / 2]);
    }
    else {
      break;
    }
  }
}

int main() {
  for(int i = 0; i < NUM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9

  printf("\n");

  for(int i = 0; i < NUM; i++) { push(A[i]); }
  while (hn > 1) { pop();  }

  for(int i = 1; i <= NUM; i++) printf("%d ", heap[i]); // 1 2 3 4 5 6 7 8 9 10
  printf("\n");
}
```

2.2.4 Quick sort

퀵 정렬(quick sort)은 분할 정복 기법을 사용하여 빠른 평균 시간 복잡도를 가지나, 최악의 경우 시간 복잡도가 높아질 수 있다.

- 시간복잡도: 평균 $O(n \log n)$, 최악 $O(n^2)$
- 공간복잡도: 평균 $O(\log n)$ (재귀 호출 스택), 최악 $O(n)$ (심하게 편향된 분할의 경우)

- C++ Skeleton Code (w/ STL)

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
constexpr int LM = 10;

int A[LM] = { 5,2,7,1,3,8,4,6,10,9 };

int main() {
  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  puts("");

  sort(A, A+LM);

  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9 10
  puts("");

  return 0;
}
```

- C++ Skeleton Code (Pure)

```cpp
// 실제 PS에서 STL 없이 정렬을 구현해야 한다면 퀵소트보다는 머지소트를 구현하는 것이 좋다
// 손구현은 피봇이나 여러 최적화 기법이 들어가있지 않기 때문에 최악의 경우 O(N*N)이 나올 수 있다
// 손구현은 참고용으로만 보는 것이 좋다
#include <bits/stdc++.h>
using namespace std;

const int LM = 100005;

int N = 10;
int A[LM] = {5,2,7,1,3,8,4,6,10,9};

void quickSort(int st, int en) {
  if(en <= st+1) return;      // base case : 수열의 길이가 1 이하이면 함수 종료
  int pivot = A[st];          // 제일 앞의 것을 pivot으로 잡는다
  int l = st+1;
  int r = en-1;
  while(1){
    while(l <= r && A[l] <= pivot) l++;
    while(l <= r && A[r] >= pivot) r--;
    if(l > r) break;
    swap(A[l], A[r]);
  }
  swap(A[st], A[r]);
  quickSort(st, r);
  quickSort(r+1, en);
}

int main() {
  quickSort(0, N);

  for(int i = 0; i < N; i++)
  printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9
  puts("");
}
```

2.2.5 Merge sort

병합 정렬(merge sort)는 분할 정복 방식을 통해 구현되며, 안정적인 정렬 방식으로 다양한 상황에서 효율적이다.

- 시간복잡도 : 평균/최악 동일하게 $O(n \log n)$
- 공간복잡도 : $O(n)$ (병합 과정에서 보조 배열 사용)
- C++ Skeleton Code (w/ STL)

```cpp
#include <cstdio>
#include <algorithm>
using namespace std;
constexpr int LM = 10;

int A[LM] = { 5,2,7,1,3,8,4,6,10,9 };

int main() {
  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  puts("");

  stable_sort(A, A+LM);

  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9 10
  puts("");

  return 0;
}
```

- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
constexpr int LM = 10;

int A[LM] = { 5,2,7,1,3,8,4,6,10,9 };
int trr[LM];

void mergeSort(int *arr, int s, int e) {
  if(s>=e) return;
  int m=(s+e)/2, i=s, j=m+1, k=s;

  mergeSort(arr,s,m), mergeSort(arr,m+1,e);

  while(i<=m && j<=e) {
    if(arr[j] < arr[i]) trr[k++] = arr[j++];
    else trr[k++] = arr[i++];
  }

  while(i<=m) trr[k++] = arr[i++];
  while(j<=e) trr[k++] = arr[j++];

  for(i=s; i<=e; i++) arr[i] = trr[i];
}

int main() {
  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 5 2 7 1 3 8 4 6 10 9
  puts("");

  mergeSort(A, 0, LM-1);

  for(int i=0; i<LM; i++) printf("%d ", A[i]); // 1 2 3 4 5 6 7 8 9 10
  puts("");

  return 0;
}
```

2.2.6 Counting sort

계수 정렬(counting sort)는 각 항목의 발생 횟수를 계산하여 정렬하는 방식으로, 작은 정수를 정렬할 때 유용하다. 비교 정렬 알고리즘의 하한은 $O(n \log n)$으로 알려져 있으나 입력된 자료의 원소가 제한적인 성질(원소의 최대값 k가 $-O(n) \sim O(n)$ 범위 내 존재해야 함)을 만족하는 경우 계수 정렬은 $O(n)$ 정렬이 가능하다. 계수 정렬은 개수를 셀 배열과 정렬된 결과가 저장될 배열이 추가로 필요한 단점이 존재한다.

- 시간복잡도: n이 작은 경우 $O(n)$
- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
constexpr int N = 10;

int A[N] = { 1,4,3,2,2,4,3,5,3,1 };
int sortedA[N];
int cnt[N];
int K = 6; // 정렬할 값의 상한

void CountingSort(int A[], int n){
  int i;
  for(i=0; i<K; i++) cnt[i]=0;        // 카운팅 배열 초기화
  for(i=0; i<n; i++) cnt[A[i]]++;     // 숫자들 카운팅
  for(i=1; i<K; i++) cnt[i] += cnt[i-1]; // 누적합 구하기
  for(i=n-1; i>=0; i--)
  sortedA[--cnt[A[i]]] = A[i];
}


int main() {

  for(int i=0; i<N; i++) printf("%d ", A[i]); // 1 4 3 2 2 4 3 5 3 1
  puts("");

  CountingSort(A, N);

  for(int i=0; i<N; i++) printf("%d ", sortedA[i]); // 1 1 2 2 3 3 3 4 4 5
  puts("");

  return 0;
}
```

2.2.7 Radix sort

기수 정렬(radix sort)는 자릿수별로 데이터를 분류하여 정렬하는 방식으로, 숫자나 문자열 정렬에 효과적이다. 기수 정렬 역시 계수 정렬과 유사하게 입력된 자료의 원소가 제한적인 성질을 만족하는 경우 더 빠른 정렬이 가능하다. 기수 정렬은 원소의 자리수가 d 자리 이하인 경우 $O(dn)$의 시간복잡도를 가진다. 또한 음수를 포함한 정수를 정렬할 수 있다.

낮은 자리부터 정렬하고 정렬된 순서를 유지하면서 보다 높은 자리를 정렬하는 과정에서 계수 정렬(counting sort)가 사용된다. 자리수 별로 정렬할 때 몫과 나머지 연산을 사용하는데 이 때 10진수 기법을 사용하면 효율성이 떨어지므로 2의 제곱수 진법을 사용한다. 일반적으로 $256(=2^8)$ 진법을 사용하며 자리수 $d=4$가 된다.

- 시간복잡도 : d 자리수 이하인 경우 $O(dn)$
- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
constexpr int N = 10;
constexpr int EXP = 8;
constexpr int MOD = (1<<EXP);    //256진법
constexpr int MASK = MOD-1;

int A[N] = { 1,4,3,2,2,4,3,5,3,1 };
int B[N];
int cnt[MOD];

void RadixSort() {
  int i, j;
  int *ap=A, *bp=B, *tp;

  for(i=0; i<32; i+=EXP) {                    // 32비트에 대하여 8비트씩 연산
    for(j=0; j<MOD; j++) cnt[j]=0;
    for(j=0; j<N; j++) cnt[(ap[j]>>i)&MASK]++; // ap[j]를 2의 i승로 나눈 몫을 256으로 나눈
    for(j=1; j<MOD; j++) cnt[j] += cnt[j-1]; // 나머지이므로 (0 - 255) 값이 나옴
    for(j=N-1; j>=0; j--)
    bp[--cnt[(ap[j]>>i)&MASK]] = ap[j];
```

```
21
22          tp=ap, ap=bp, bp=tp; //  - 128) * 64 + (‘ - 128) 끌 - 128) * 64 + (‘ - 128) 뮘 swap
23        }
24      }
25
26
27      int main() {
28        for(int i=0; i<N; i++) printf("%d ", A[i]); // 1 4 3 2 2 4 3 5 3 1
29        puts("");
30
31        RadixSort();
32
33        for(int i=0; i<N; i++) printf("%d ", A[i]); // 1 1 2 2 3 3 3 4 4 5
34        puts("");
35
36        return 0;
37      }
```

2.3 Search

2.3.1 Binary search

이진 탐색(binary search)는 정렬된 리스트에서 중간값을 기준으로 탐색 범위를 반으로 줄여가며 원하는 값을 찾는 방법이다. 이진 탐색을 수행하기 전 데이터는 오름차순으로 정렬되어 있어야 한다.

- 시간복잡도: $O(\log n)$
- C++ Skeleton Code (w/ STL)

```
1     #include <cstdio>
2     #include <algorithm>
3
4     constexpr int LM = 1005;
5
6     int A[LM];
7     int N;
8
9     int main() {
10      N = 10;
11      A[0] = 5; A[1] = 2; A[2] = 7; A[3] = 1; A[4] = 3;
12      A[5] = 9; A[6] = 4; A[7] = 6; A[8] = 10; A[9] = 9; // 5 2 7 1 3 9 4 6 10 9
13
14      std::sort(A, A + N);                          // 1 2 3 4 5 6 7 8 9 10
15
16      int ret = std::binary_search(A, A+N, 7);
17
18      if(ret == 1) printf("found\n");
19      else printf("not found\n");
20    }
```

- C++ Skeleton Code (Pure)

```
1     #include <cstdio>
2     #include <algorithm>
3
4     constexpr int LM = 1005;
5
6     int A[LM];
7     int N;
8
9     int binarySearch(int target) {
10      int l = 0;
11      int r = N-1;
12
13      while (l <= r) {
14        int m = (l+r) / 2;
15
16        if (A[m] == target) return 1; // 찾은 경우
```

```
17
18          if (A[m] < target) l = m + 1;
19          else r = m - 1;
20        }
21
22        return 0; // 찾지 못한 경우
23      }
24
25      int main() {
26        N = 10;
27        A[0] = 5; A[1] = 2; A[2] = 7; A[3] = 1; A[4] = 3;
28        A[5] = 9; A[6] = 4; A[7] = 6; A[8] = 10; A[9] = 9; // 5 2 7 1 3 9 4 6 10 9
29
30        std::sort(A, A + N);                          // 1 2 3 4 5 6 7 8 9 10
31
32        int ret = binarySearch(7);
33
34        if(ret == 1) printf("found\n");
35        else printf("not found\n");
36      }
```

2.3.2 Parametric search

매개변수 탐색(parametric search)는 조건을 만족하는 최소/최대값을 구하는 최적화 문제를 결정 문제로 변환하여 이분 탐색을 수행하는 방법을 말한다.

- BOJ 1654 랜선 자르기
- (최적화 문제) N개를 만들 수 있는 랜선의 최대 길이 X는? (Maximize X, subject to N)
- (결정 문제) 랜선의 길이가 X일 때 랜선이 N개 이상인가? (Yes or No)

```
1     // 랜선 자르기 boj.kr/1654
2     #include <bits/stdc++.h>
3     using namespace std;
4     using ll = long long;
5
6     const int LM = 10005;
7
8     int N,K;
9     int A[LM];
10
11    // Parametric search
12    bool solve(ll x){
13      ll cur = 0;
14      for(int i=0;i<K;i++)
15      cur += A[i]/x;
16      return cur >= N;
17    }
18
19    int main(int argc, char **argv){
20      scanf("%d %d",&K,&N);
21      for(int i=0;i<K;i++)
22      scanf("%d", &A[i]);
23
24      ll st = 1;
25      ll en = 0x7fffffff; // 2^31 - 1
26
27      while(st < en) {
28        ll m = (st+en+1)/2;
29        if(solve(m)) st = m;
30        else en = m-1;
31      }
32
33      printf("%lld\n", st);
34    }
```

2.3.3 DFS

깊이 우선 탐색(Depth First Search, DFS)은 그래프의 깊은 부분을 우선적으로 탐색하는 방식이다.

- 시간복잡도: $O(V+E)$
- 공간복잡도: $O(V)$
- C++ Skeleton Code (w/ STL)

```cpp
#include <cstdio>
#include <cstring>
#include <vector>
#include <stack>
using namespace std;

constexpr int LM = 1005;

vector<vector<int>> A;
int visited[LM];

void DFS(int start) {
  stack<int> st;
  st.push(start);
  visited[start] = 1;

  while (!st.empty()) {
    int current = st.top();
    st.pop();

    printf("%d ", current);

    // 현재 노드에 인접한 노드를 스택에 추가
    // 인접한 노드를 거꾸로 탐색하여 스택에 넣어줌으로써, 재귀 버전의 탐색 순서를 유지
    for (int i = A[current].size() - 1; i >= 0; i--) {
      int next = A[current][i];
      if (visited[next]) continue;
      visited[next] = 1;
      st.push(next);
    }
  }
}

int main() {
  A = {         // 그래프 표현 (인접 리스트)
    {1, 2},   // 정점 0에 인접한 정점: 1, 2
    {0, 3, 4}, // 정점 1에 인접한 정점: 0, 3, 4
    {0, 4},   // 정점 2에 인접한 정점: 0, 4
    {1, 5},   //정점 3에 인접한 정점: 1, 5
    {1, 2},   // 정점 4에 인접한 정점: 1, 2
    {3}       //정점 5에 인접한 정점: 3
  };

  memset(visited, 0, sizeof(visited));

  DFS(0); // 시작 정점을 스택에 넣고 DFS 시작

  return 0;
}
```

- C++ Skeleton Code (Pure)

```cpp
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
constexpr int LM = 1005;

vector<vector<int>> A;
int visited[LM];
```

```
9
10      void DFS(int s) {
11        if(visited[s]) return; // base condition
12
13        visited[s] = 1;
14        printf("%d ", s);
15
16        for(int i = 0; i < A[s].size(); i++) {
17          int next = A[s][i];
18          if(!visited[next]) DFS(next);
19        }
20      }
21
22      int main() {
23        A = {         // 그래프 표현 (인접 리스트)
24          {1, 2},   // 정점 0에 인접한 정점: 1, 2
25          {0, 3, 4}, //정점 1에 인접한 정점: 0, 3, 4
26          {0, 4},   // 정점 2에 인접한 정점: 0, 4
27          {1, 5},   // 정점 3에 인접한 정점: 1, 5
28          {1, 2},   // 정점 4에 인접한 정점: 1, 2
29          {3}       // 정점 5에 인접한 정점: 3
30        };
31
32        memset(visited, 0, sizeof(visited));
33
34        DFS(0);     // 0 1 3 5 4 2
35
36        return 0;
37      }
```

2.3.4 BFS

너비 우선 탐색(Breadth First Search, BFS)은 가까운 노드를 먼저 탐색하며, 레벨 별로 탐색하는 방식이다.

- 시간복잡도: $O(V + E)$
- 공간복잡도: $O(V)$
- C++ Skeleton Code (w/ STL)

```
1       #include <cstdio>
2       #include <cstring>
3       #include <vector>
4       #include <queue>
5       using namespace std;
6       constexpr int LM = 1005;
7
8       vector<vector<int>> A;
9       queue<int> q;
10      int visited[LM];
11
12      void BFS(int s) {
13        q.push(s);
14        visited[s] = 1; // 시작 정점을 큐에 넣고 방문 처리
15
16        while (!q.empty()) {
17          int cur = q.front();
18          q.pop();
19
20          printf("%d ", cur);
21
22          for (int i = 0; i < A[cur].size(); i++) { // 현재 정점에 인접한 모든 정점을 탐색
23            int next = A[cur][i];
24            if (visited[next]) continue; // 이미 방문한 정점은 스킵
25
26            q.push(next);               // 아직 방문하지 않은 정점이라면 큐에 넣고
27            visited[next] = 1;          //방문 처리
28          }
29        }
```

```
  puts("");
}

int main() {
  A = {         // 그래프 표현 (인접 리스트)
    {1, 2},   // 정점 0에 인접한 정점: 1, 2
    {0, 3, 4}, // 정점 1에 인접한 정점: 0, 3, 4
    {0, 4},   // 정점 2에 인접한 정점: 0, 4
    {1, 5},   // 정점 3에 인접한 정점: 1, 5
    {1, 2},   // 정점 4에 인접한 정점: 1, 2
    {3}       // 정점 5에 인접한 정점: 3
  };

  memset(visited,0, sizeof(visited));

  BFS(0);    // 0 1 2 3 4 5

  return 0;
}
```

- C++ Skeleton Code (Pure)

```
#include <cstdio>
#include <cstring>
#include <vector>
using namespace std;
constexpr int LM = 1005;

vector<vector<int>> A;

int que[LM*LM];
int visited[LM];
int fr, re;

void BFS(int s) {
  fr = re = 0;

  que[re++] = s;
  visited[s] = 1; //시작 정점을 큐에 넣고 방문 처리

  while (fr < re) {
    int cur = que[fr++];

    printf("%d ", cur);

    for (int i = 0; i < A[cur].size(); i++) { // 현재 정점에 인접한 모든 정점을 탐색
      int next = A[cur][i];
      if (visited[next]) continue; // 이미 방문한 정점은 스킵

      que[re++] = next;              // 아직 방문하지 않은 정점이라면 큐에 넣고
      visited[next] = 1;          // 방문 처리
    }
  }
  puts("");
}

int main() {
  A = {         // 그래프 표현 (인접 리스트)
    {1, 2},   // 정점 0에 인접한 정점: 1, 2
    {0, 3, 4}, // 정점 1에 인접한 정점: 0, 3, 4
    {0, 4},   // 정점 2에 인접한 정점: 0, 4
    {1, 5},   // 정점 3에 인접한 정점: 1, 5
    {1, 2},   // 정점 4에 인접한 정점: 1, 2
    {3}       // 정점 5에 인접한 정점: 3
  };

  memset(visited,0, sizeof(visited));

  BFS(0);    // 0 1 2 3 4 5

```

```
  return 0;
}
```

2.4 Two pointers

투 포인터(two pointers) 알고리즘은 배열에서 원래 이중 for문으로 $O(n^2)$에 처리되는 작업을 2개의 포인터의 움직임으로 $O(n)$에 해결하는 알고리즘을 말한다.

- 시간복잡도: $O(n)$
- C++ Skeleton Code (Pure)

```
// 수 고르기 boj.kr/2230
#include <bits/stdc++.h>
using namespace std;

constexpr int LM = 100'005;
constexpr int INF = 0x7fffffff;

int N,M;
int A[LM];

int main(int argc, char **argv){
  scanf("%d%d",&N, &M);

  for(int i=0; i<N; i++){
    scanf("%d", &A[i]);
  }

  sort(A, A+N);

  int ans = INF;

  // Two pointers
  int en = 0;
  for(int st=0; st<N; st++){
    while(en < N && A[en] - A[st] < M) en++;
    if(en == N) break; // en이 범위를 벗어날 시 종료
    ans = min(ans, A[en] - A[st]);
  }

  printf("%d\n", ans);
}
```

2.5 Divide and conquer

분할정복(divide and conquer)은 주어진 문제를 둘 이상의 부분 문제로 나눈 뒤 각 문제에 대한 답을 재귀 호출을 이용해 계산하고 각 부분 문제의 답으로 전체 문제의 답을 계산하는 알고리즘을 말한다. 분할 정복이 재귀 호출과 다른 점은 문제를 한 조각과 나머지 전체로 나누는 대신 거의 같은 크기의 부분 문제로 나누는 것이다. 분할정복은 많은 경우 같은 작업을 더 빠르게 처리할 수 있다.

- 1+2+...+n까지의 합을 정복분할로 푸는 예제 (C++)

```
#include <bits/stdc++.h>
using ll = long long;

// 시간복잡도 O(log n)
ll fastSum(int n){
  if(n==1) return 1;

  if(n%2 == 1) return fastSum(n-1) + n;
  else return 2*fastSum(n/2) + (n/2)*(n/2);
}

int main(int argc, char **argv){
  printf("%lld\n", fastSum(9651)); // 46575726
```

```
}
```

- 정방행렬의 거듭제곱을 정복분할로 푸는 예제 (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

class SquareMatrix {
  public:
  vector<vector<long long>> mat;
  int n;
  SquareMatrix(int n) : n(n), mat(n, vector<long long>(n, 0)) {}
  SquareMatrix(const vector<vector<long long>>& arr)
  : n((int)arr.size()), mat(arr) {}

  int size() const { return n; } // 행렬 크기 반환

  SquareMatrix operator*(const SquareMatrix &r) const {
    SquareMatrix ret(n);
    for(int i=0; i<n; i++){
      for(int j=0; j<n; j++){
        long long sum = 0;
        for(int k=0; k<n; k++){
          sum += mat[i][k] * r.mat[k][j];
        }
        ret.mat[i][j] = sum;
      }
    }
    return ret;
  }
};

SquareMatrix identity(int n){
  SquareMatrix I(n);
  for(int i=0; i<n; i++){
    I.mat[i][i] = 1;
  }
  return I;
}

// 분할정복으로 행렬 거듭제곱
// 시간복잡도: O(n^3 log m), n: 행렬 크기, m: 거듭제곱 횟수
// 분할정복 안 쓴 경우 O(n^3 m)
SquareMatrix pow(const SquareMatrix& A, int m) {
  if(m == 0) return identity(A.size());
  if(m % 2 > 0) return pow(A, m - 1) * A;
  SquareMatrix half = pow(A, m / 2);
  return half * half;
}

int main(){
  SquareMatrix M({{1,2},{3,4}}); // 2x2행렬 예시
  int exponent = 5;

  SquareMatrix result = pow(M, exponent);

  for(int i=0; i<result.size(); i++){
    for(int j=0; j<result.size(); j++){
      cout << result.mat[i][j] << " ";
    }
    cout << "\n";
  }
}
```

- 두 큰 정수의 곱셈을 정복분할로 푸는 예제 (C++) (a.k.a 카라츠바 알고리즘)

```cpp
#include <bits/stdc++.h>
using namespace std;

```

```
4       // 카라츠바 시간복잡도: O(n^log3) ~= O(n^1.585)
5       // 단순히 두 큰 수를 곱하는 O(n^2)보다 훨씬 적은 곱셈을 필요로 한다
6
7       // num[]의 자리수 올림을 처리한다
8       void normalize(vector<int>& num){
9         num.push_back(0);
10
11        //자리수 올림 처리
12        for(int i=0; i<num.size()-1; i++){
13          if(num[i] < 0) {
14            int borrow = (abs(num[i]) + 9) / 10;
15            num[i+1] -= borrow;
16            num[i] += borrow * 10;
17          }
18          else {
19            num[i+1] += num[i] / 10;
20            num[i] %= 10;
21          }
22        }
23
24        while(num.size() > 1 && num.back() == 0)
25        num.pop_back();
26      }
27
28      // 두 긴 자연수의 곱을 반환한다
29      //각 배열에는 각 수의 자리수가 1의 자리에서부터 시작해 저장되어있다
30      vector<int> multiply(const vector<int>& a, const vector<int>& b) {
31        vector<int> c(a.size() + b.size() + 1, 0);
32        for(int i=0; i<a.size(); i++){
33          for(int j=0; j<b.size(); j++){
34            c[i+j] += a[i] * b[j];
35          }
36        }
37        normalize(c);
38        return c;
39      }
40
41      // a += b * (10^k)
42      void addTo(vector<int>& a, const vector<int>& b, int k) {
43        // Ensure a has enough size to accommodate b shifted by k
44        int sizeNeeded = max((int)a.size(), (int)(b.size() + k));
45        if(a.size() < sizeNeeded) {
46          a.resize(sizeNeeded, 0);
47        }
48        // Digit-wise addition
49        for(int i=0; i<b.size(); i++){
50          a[i + k] += b[i];
51        }
52        normalize(a);
53      }
54
55      // a -= b
56      void subFrom(vector<int>& a, const vector<int>& b) {
57        // Subtract b from a
58        for(int i=0; i<b.size(); i++){
59          a[i] -= b[i];
60        }
61        normalize(a);
62      }
63
64      vector<int> karatsuba(const vector<int>& a, const vector<int>& b) {
65        int an = a.size(), bn = b.size();
66        // a가 b보다 작으면 swap
67        if(an < bn) return karatsuba(b,a);
68
69        if(an == 0 || bn == 0)
70        return vector<int>();
71
72        // 작은 크기에서는 일반 곱으로 계산
73        if(an <= 50)
```

```
    return multiply(a,b);

    int half = an / 2;

    // a, b를 절반으로 분할
    vector<int> a0(a.begin(), a.begin() + half);
    vector<int> a1(a.begin() + half, a.end());
    vector<int> b0(b.begin(), b.begin() + min<int>(b.size(), half));
    vector<int> b1(b.begin() + min<int>(b.size(), half), b.end());

    // 재귀적으로 계산
    vector<int> z2 = karatsuba(a1, b1);
    vector<int> z0 = karatsuba(a0, b0);

    // (a0 + a1)*(b0 + b1)를 계산
    addTo(a0, a1, 0); // a0 = a0 + a1
    addTo(b0, b1, 0); // b0 = b0 + b1
    vector<int> z1 = karatsuba(a0, b0);

    // z1 = z1 - z0 - z2
    subFrom(z1, z0);
    subFrom(z1, z2);

    // 결과를 ret에 합치기
    vector<int> ret;
    // 시작은 0으로 아무것도 없는 상태
    addTo(ret, z0, 0);
    addTo(ret, z1, half);
    addTo(ret, z2, half + half);

    return ret;
}

int main(){
    string s1, s2;
    s1 = "1234";
    s2 = "5678";

    vector<int> a, b;
    for(int i = s1.size()-1; i >= 0; i--){
        a.push_back(s1[i] - '0');
    }
    for(int i = s2.size()-1; i >= 0; i--){
        b.push_back(s2[i] - '0');
    }

    // Karatsuba로 곱셈
    vector<int> result = karatsuba(a, b);

    // Leading zero 제거 (최소 한 자리는 남겨둠)
    while(result.size() > 1 && result.back() == 0) {
        result.pop_back();
    }

    // 정상적인(가장 큰 자리부터) 순서로 출력
    for(int i = result.size()-1; i >= 0; i--){
        printf("%d", result[i]);
    }
    puts("");
}
```

2.6 Dynamic programming

동적계획법(dynamic programming)은 복잡한 문제를 작은 하위 문제로 나누어 해결하고, 이 결과를 저장하여 재사용함으로써 전체 문제의 효율적인 해결을 가능하게 한다. 최적 부분 구조와 중복되는 하위 문제를 가진 경우에 유용하다.

- 최대 증가 부분 수열(LIS)을 DP로 푸는 예제 (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

// 시간복잡도: O(n^2)
int N;
int dp[100];
vector<int> A;

int lis(int start) {
  int &ret = dp[start];
  if(ret != -1) return ret;

  // 항상 A[start]는 있기 때문에 길이는 최하 1
  ret = 1;
  for(int next = start+1; next<N; next++){
    if(A[start] < A[next])
    ret = max(ret, lis(next)+1);
  }

  return ret;
}

int main(int argc, char **argv){
  N = 8;

  memset(dp, -1, sizeof(dp));

  A.push_back(5);
  A.push_back(6);
  A.push_back(7);
  A.push_back(8);
  A.push_back(1);
  A.push_back(2);
  A.push_back(3);
  A.push_back(4); // 5 6 7 8 1 2 3 4

  int ans=0;
  for(int i=0; i<N; i++)
  ans = max(ans, lis(i));

  printf("%d\n", ans); // 4

}
```

- 피보나치 수열을 DP로 구하는 예제 (C++)

```cpp
#include <cstdio>

// recursive : O(1.618^n)
// dp : O(n)
int fibonacci(int n){
  int f[50];
  f[0] = f[1] = 1;
  for(int i=2; i<=n; i++)
  f[i] = f[i-1] + f[i-2];
  return f[n];
}

int main() {
  printf("%d\n", fibonacci(20)); // 10946
}
```

2.7 Greedy

탐욕(greedy) 알고리즘은 매 선택에서 지역적으로 최적인 선택을 함으로써 최종적인 해답의 최적성을 보장하는 알고리즘이다. 문제에 따라 최적해를 보장하지 않을 수도 있다.

2.8 Dijkstra

다익스트라(Dijkstra) 알고리즘은 가중치가 있는 그래프에서 두 정점 사이의 최단 경로를 찾는 알고리즘이다. 음수 가중치를 가진 간선이 없을 때 사용할 수 있다. 알고리즘은 출발점에서 각 정점까지의 최단 거리를 저장하면서, 가장 가까운 정점을 선택하고, 이 정점을 통해 다른 정점으로 가는 거리를 업데이트하는 과정을 반복한다.

- C++ Skeleton Code (Pure) (시간복잡도: $O(V^2+E)$: V가 20,000개가 넘으면 PS 알고리즘에서 보통 TLE 발생)

```cpp
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;

#define X first
#define Y second

constexpr int LM = 20005;
constexpr int INF = 0x3f3f3f3f;

vector<pii> adj[LM];

bool fix[LM];
int d[LM];
int V, E, K;

void dijkstraNaive(int s) {
  fill(d, d + V + 1, INF); // 최단거리 테이블 초기화
  d[s] = 0;

  while (1) {
    int idx = -1;
    for (int i = 1; i <= V; i++) {
      if (fix[i]) continue;
      if (idx == -1) idx = i;
      else if (d[i] < d[idx]) idx = i;
    }
    if (idx == -1 || d[idx] == INF) break; // 더 이상 선택할 정점이 없으면

    fix[idx] = 1; // 정점 idx 고정

    for(auto next : adj[idx])
    d[next.Y] = min(d[next.Y], d[idx] + next.X);
  }
}

int main() {
  V = 5;  // 노드
  E = 6;  // 엣지
  K = 1;  // 시작 정점 번호

  // u, v, w
  vector<tuple<int, int, int>> edges = {
    {5, 1, 1},
    {1, 2, 2},
    {1, 3, 3},
    {2, 3, 4},
    {2, 4, 5},
    {3, 4, 6}
  };

  for (auto [u, v, w] : edges) {
    adj[u].push_back({ w, v });
  }

  dijkstraNaive(K);

  for (int i = 1; i <= V; i++) {
    if (d[i] == INF) printf("INF\n");
```

```
      else printf("%d\n", d[i]);
    }
    // 0
    // 2
    // 3
    // 7
    // INF
}
```

- C++ Skeleton Code (w/ STL) (시간복잡도: $O(E \log V)$: 일반적으로 PS에서 자주 사용되는 방식)

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;

#define X first
#define Y second

constexpr int LM = 20005;
constexpr int INF = 0x3f3f3f3f;

vector<pii> adj[LM];
priority_queue<pii, vector<pii>, greater<pii>> pq;

bool fix[LM];
int d[LM];
int V, E, K;

void dijkstra(int s) {
  fill(d, d + V + 1, INF);
  d[s] = 0;

  pq.push({ d[s], s });

  while (!pq.empty()) {
    auto cur = pq.top(); pq.pop();

    if (d[cur.Y] != cur.X) continue;

    for (auto next : adj[cur.Y]) {
      if (d[next.Y] <= d[cur.Y] + next.X) continue;
      d[next.Y] = d[cur.Y] + next.X;
      pq.push({ d[next.Y], next.Y });
    }
  }
}

int main() {
  V = 5;  // 노드
  E = 6;  // 엣지
  K = 1;  // 시작 정점 번호

  // u, v, w
  vector<tuple<int, int, int>> edges = {
    {5, 1, 1},
    {1, 2, 2},
    {1, 3, 3},
    {2, 3, 4},
    {2, 4, 5},
    {3, 4, 6}
  };

  for (auto [u, v, w] : edges) {
    adj[u].push_back({ w, v });
  }

  dijkstra(K);

  for (int i = 1; i <= V; i++) {
    if (d[i] == INF) printf("INF\n");
```

```
      else printf("%d\n", d[i]);
    }
    // 0
    // 2
    // 3
    // 7
    // INF
}
```

- C++ Skeleton Code (w/ STL) (+ 경로 복원) (시간복잡도: $O(E \log V)$: 위와 동일)

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;

#define X first
#define Y second

const int LM = 1005;
const int INF = 1e9 + 10;

vector<pii> adj[LM];

int v, e, st, en;
int d[LM];
int pre[LM];

int main() {
  v = 5;
  e = 8;

  // a,b,c (a,b : node) (c : cost)
  vector<tuple<int, int, int>> edges = {
    {1, 2, 2}, {1, 3, 3}, {1, 4, 1}, {1, 5, 10},
    {2, 4, 2}, {3, 4, 1}, {3, 5, 1}, {4, 5, 3}
  };

  st = 1;  // start
  en = 5;  // end

  fill(d, d + v + 1, INF);

  for (auto [u, v, w] : edges) {
    adj[u].push_back({w, v});
  }

  // Dijkstra
  priority_queue<pii, vector<pii>, greater<pii>> pq;
  d[st] = 0;
  pq.push({d[st], st});

  while (!pq.empty()) {
    auto cur = pq.top(); pq.pop();

    if (d[cur.Y] != cur.X) continue;
    for (auto next : adj[cur.Y]) {
      if (d[next.Y] <= d[cur.Y] + next.X) continue;

      d[next.Y] = d[cur.Y] + next.X;
      pq.push( { d[next.Y], next.Y } );
      pre[next.Y] = cur.Y;        // 경로 복원
    }
  }

  //도착 도시까지 최소 비용
  printf("%d\n", d[en]);

  //경로 복원
  vector<int> path;
  int cur = en;
```

```
    while (cur != st) {
      path.push_back(cur);
      cur = pre[cur];
    }

    path.push_back(cur);
    reverse(path.begin(), path.end());
    printf("%d\n", path.size());       // 도시 개수
    for (auto x : path) printf("%d ", x); // 최소 비용을 갖는 경로
  }
```

2.9 Floyd-Warshall

플로이드-워셜(Floyd-Warshall) 알고리즘은 모든 정점 쌍 간의 최단 경로를 찾는 알고리즘이다. 이 알고리즘은 동적 프로그래밍을 기반으로 하며, 그래프의 모든 정점을 거쳐 가는 경로를 고려하여 최단 거리를 계산한다. 이는 각 정점을 거쳐 가는 모든 경로의 최단 거리를 행렬로 저장하고 업데이트하는 방식으로 작동한다.

- 시간복잡도: $O(V^3)$
- 공간복잡도: $O(V^2)$
- C++ Skeleton Code (Pure)

```
#include <bits/stdc++.h>
using namespace std;

const int LM = 105;
const int INF = 0x3f3f3f3f;

int d[LM][LM];
int n, m;

// Floyd-Warshall
void floyd() {
  for (int i = 1; i <= n; i++)
  d[i][i] = 0;

  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
      }
    }
  }
}

int main() {
  n = 5;   // node
  m = 14;  // edge

  // a,b,c (a,b : node) (c : cost)
  vector<tuple<int, int, int>> edges = {
    {1, 2, 2}, {1, 3, 3}, {1, 4, 1}, {1, 5, 10},
    {2, 4, 2}, {3, 4, 1}, {3, 5, 1}, {4, 5, 3},
    {3, 5, 10}, {3, 1, 8}, {1, 4, 2}, {5, 1, 7},
    {3, 4, 2}, {5, 2, 4}
  };

  for (int i = 1; i <= n; i++) {
    fill(d[i], d[i] + n + 1, INF);
  }

  for (auto [a, b, c] : edges) {
    d[a][b] = min(d[a][b], c);
  }

```

```
    floyd();

    // 최단 거리 테이블
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        if (d[i][j] == INF) printf("0 ");
        else printf("%d ", d[i][j]);
      }
      puts("");
    }
  }
```

- C++ Skeleton Code (Pure) (using dp) (+경로 복원)

```
#include <bits/stdc++.h>
using namespace std;

const int LM = 105;
const int INF = 0x3f3f3f3f;

int d[LM][LM];
int nxt[LM][LM];
int n, m;

// Floyd-Warshall
void floyd(){
  for (int i = 1; i <= n; i++)
  d[i][i] = 0;

  for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
      for (int j = 1; j <= n; j++) {
        if (d[i][k] + d[k][j] < d[i][j]) {
          d[i][j] = d[i][k] + d[k][j];
          nxt[i][j] = nxt[i][k];
        }
      }
    }
  }
}

//경로 복원
void restorePath() {
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (d[i][j] == 0 || d[i][j] == INF) {
        printf("0\n");
        continue;
      }

      vector<int> path;
      int st = i;
      while (st != j) {
        path.push_back(st);
        st = nxt[st][j];
      }
      path.push_back(j);
      printf("%d ", (int)path.size());
      for (auto x : path) printf("%d ", x);
      puts("");
    }
  }
}

int main() {
  n = 5;  // node
  m = 14; // edge

  // a,b,c (a,b : node) (c : cost)
  vector<tuple<int, int, int>> edges = {
```

```
    {1, 2, 2}, {1, 3, 3}, {1, 4, 1}, {1, 5, 10},
    {2, 4, 2}, {3, 4, 1}, {3, 5, 1}, {4, 5, 3},
    {3, 5, 10}, {3, 1, 8}, {1, 4, 2}, {5, 1, 7},
    {3, 4, 2}, {5, 2, 4}
  };

  for (int i = 1; i <= n; i++) {
    fill(d[i], d[i] + n + 1, INF);
  }

  for (auto [a, b, c] : edges) {
    d[a][b] = min(d[a][b], c);
    nxt[a][b] = b;
  }

  floyd();

  // 최단 거리 테이블
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      if (d[i][j] == INF) printf("0 ");
      else printf("%d ", d[i][j]);
    }
    puts("");
  }

  restorePath();
}
```

2.10 Minimum spanning tree

최소 비용 신장트리(minimum spanning tree, MST)는 그래프의 모든 노드를 최소의 연결 비용으로 연결하는 부분 그래프(트리)이다. 프림(Prim) 알고리즘과 크루스칼(Kruskal) 알고리즘은 MST를 찾는 데 널리 사용되는 알고리즘이다. 이들은 각각 탐욕적 방법을 사용하여 전체 그래프에서 최소 비용의 에지들을 선택하여 MST를 구성한다.

- 크루스칼 알고리즘 (C++) (using Union-find) (시간복잡도: $O(E \log E)$)

```
#include <bits/stdc++.h>
using namespace std;
using tiii = tuple<int,int,int>;

vector<int> p(10005,-1);

int find(int x){
  if(p[x] < 0) return x;
  return p[x] = find(p[x]);
}

bool isDiffGroup(int a, int b){
  a = find(a); b = find(b);
  if(a == b) return 0;

  if(p[a] == p[b]) p[a]--;
  if(p[a] < p[b]) p[b] = a;
  else p[a] = b;

  return 1;
}

int main(void) {
  int v = 3, e = 3;

  // a,b,c (a,b : node) (c : cost)
  tiii edge[] = {
    {1, 2, 1},
    {2, 3, 2},
    {1, 3, 3}
```

```
    };

    sort(edge, edge + e);

    int cnt = 0;
    int ans = 0;

    // Kruskal algorithm (using isDiffGroup-find)
    for(int i = 0; i < e; i++){
      int a, b, cost;
      tie(cost, a, b) = edge[i];

      if(!isDiffGroup(a, b)) continue;

      ans += cost; // MST 가중치의 합
      cnt++;
      if(cnt == v-1) break;
    }

    printf("%d\n", ans); // 3
}
```

- 프림 알고리즘 (C++) (using pq) (시간복잡도: $O(E \log V)$)

```
#include <bits/stdc++.h>
#define X first
#define Y second
using namespace std;
using pii = pair<int, int>;
using tiii = tuple<int, int, int>;

constexpr int LM = 10005;

vector<pii> adj[LM]; // adj[i] = { cost , vertex_id }

bool chk[LM]; // chk[i] : i번째 정점이 MST에 속해있는가?
int cnt = 0; //현재 선택된 간선의 수

int main() {
  int v = 3, e = 3;

  // a,b,c (a,b : node) (c : cost)
  vector<tuple<int, int, int>> edges = {
    {1, 2, 1},
    {2, 3, 2},
    {1, 3, 3}
  };

  for (auto [a, b, cost] : edges) {
    adj[a].push_back({ cost, b });
    adj[b].push_back({ cost, a });
  }

  // Prim algorithm
  priority_queue<tiii, vector<tiii>, greater<tiii>> pq; // pq[i] = { cost, vertex_id1,
      vertex_id2 }

  chk[1] = 1;

  for (auto nxt : adj[1]) {
    pq.push({ nxt.X, 1, nxt.Y });
  }

  int ans = 0;

  while (cnt < v - 1) {
    int cost, a, b;
    tie(cost, a, b) = pq.top(); pq.pop();

    if (chk[b]) continue;
```

```
46
47          ans += cost; // MST의 가중치 합
48          chk[b] = 1;
49          cnt++;
50
51          for (auto nxt : adj[b]) {
52            if (!chk[nxt.Y])
53            pq.push({ nxt.X, b, nxt.Y });
54          }
55        }
56
57
58        printf("%d\n", ans); // 3
59      }
```

2.11 Topological sort

위상 정렬(topological sorting)은 방향 비순환 그래프(DAG, Directed Acyclic Graph)의 모든 노드를 방향성에 위배되지 않도록 순서대로 나열하는 정렬 방법이다. 이는 주로 의존성이 있는 여러 작업을 수행해야 할 때 그 순서를 결정하는 데 사용된다.

- C++ Skeleton Code (Pure)

```
1   #include <bits/stdc++.h>
2   using namespace std;
3
4   const int LM = 32005;
5
6   vector<int> adj[LM];
7   int deg[LM];
8   int N;
9
10  int main() {
11    // Nodes
12    N = 4;
13
14    // Edges
15    adj[4].push_back(2);
16    deg[2]++;
17    adj[3].push_back(1);
18    deg[1]++;
19
20    // Topological sort
21    queue<int> q;
22    vector<int> result;
23
24    for (int i = 1; i <= N; i++)
25    if (deg[i] == 0) q.push(i);
26
27    while (!q.empty()) {
28      int cur = q.front(); q.pop();
29      result.push_back(cur);
30
31      for (int nxt : adj[cur]) {
32        deg[nxt]--;
33        if (deg[nxt] == 0) q.push(nxt);
34      }
35    }
36
37    for (auto x : result) printf("%d ", x); // 3 4 1 2
38  }
```

2.12 KMP

KMP(Knuth-Morris-Pratt) 알고리즘은 문자열 검색을 효율적으로 수행하는 알고리즘으로, 패턴 내에서 접두사와 접미사의 일치 정보를 활용하여 불필요한 비교를 줄인다. 전처리 연산을 통해 패턴의 "접두사 배

열(longest prefix suffix, LPS 배열 또는 실패 함수)"을 생성하고, "탐색(search)" 연산을 수행할 때 이를 활용하여 중복된 비교 없이 빠르게 문자열 내에서 패턴을 찾는다. 일반적인 브루트포스 검색보다 빠르며, 대량의 텍스트 검색에서 효과적이다. KMP는 문자열 검색, DNA 서열 분석, 편집 거리 계산 등의 문제에서 활용된다.

- $|N|$: 전체 문자열 N의 길이
- $|M|$: 검색할 패턴 M의 길이
- 시간복잡도: $O(|N| + |M|)$ (브루트포스 기반 문자열 매칭 : $O(|N| \times |M|)$)
- C++ Skeleton Code (Pure)

```cpp
// 부분 문자열 boj.kr/16916
#include <bits/stdc++.h>
using namespace std;

string s, p;

// longest prefix suffix,LPS 배열(실패 함수) 구하기
vector<int> failure( string& s ) {
  vector<int> f( s.size() );
  int j = 0;

  for ( int i = 1; i < s.size(); i++ ) {
    while ( j > 0 && s[i] != s[j] ) j = f[j - 1];
    if ( s[i] == s[j] ) f[i] = ++j;
  }
  return f;
}

void kmpSearch(){
  vector<int> f = failure( p );
  int j = 0;
  for ( int i = 0; i < s.size(); i++ ) {
    while ( j > 0 && s[i] != p[j] ) j = f[j - 1];
    if ( s[i] == p[j] ) j++;
    if ( j == p.size() ) {
      printf( "1\n" );
      return;
    }
  }
  printf( "0\n" );
}

int main( int argc, char** argv ) {
  cin >> s >> p;
  // s : baekjoon
  // p : aek

  kmpSearch(); // 1
}
```

2.13 Manber-Myers (Suffix array)

맨버-마이어스(Manber-Myers) 알고리즘은 문자열의 접미사 배열(Suffix array)을 효율적으로 구성하는 대표적인 방법이다. 이 알고리즘은 처음에 각 접미사를 한 글자씩 비교하여 정렬한 뒤, 비교 길이를 두 배씩 늘려가며 정렬을 반복한다. 이전 단계의 순위를 활용해 빠르게 다음 정렬 기준을 계산할 수 있기 때문에 매우 효율적인 알고리즘이다. 구현이 비교적 간단한 편이며, 문자열 검색, 중복 문자열 처리, 최장 공통 접두사(LCP) 계산 등 다양한 문자열 알고리즘의 기반이 되는 자료구조로 널리 사용된다.

- $|N|$: 문자열 N의 길이
- 시간복잡도: $O(|N| \log |N|)$
- C++ Skeleton Code (Pure)

```cpp
#include <bits/stdc++.h>
using namespace std;

// Manber-Myers
vector<int> buildSuffixArray(const string& s) {
  int n = s.size();
  vector<int> perm(n), group(n), ngroup(n);

  // 초기 정렬: 문자 기준
  for (int i = 0; i < n; ++i) {
    perm[i] = i;
    group[i] = s[i];
  }

  for (int t = 1;; t *= 2) {
    auto cmp = [&](int i, int j) -> bool {
      if (group[i] != group[j]) return group[i] < group[j];
      int ri = (i+t < n) ? group[i+t] : -1;
      int rj = (j+t < n) ? group[j+t] : -1;
      return ri < rj;
    };

    sort(perm.begin(), perm.end(), cmp);

    ngroup[perm[0]] = 0;
    for (int i = 1; i < n; ++i)
    ngroup[perm[i]] = ngroup[perm[i-1]] + cmp(perm[i-1], perm[i]);

    group = ngroup;
    if (group[perm[n-1]] == n-1) break; // 모든 순위가 유일하면 종료
  }

  return perm;
}

int main() {
  string s = "banana";
  vector<int> perm = buildSuffixArray(s);

  cout << "Suffix array:\n";
  for (int i : perm)
  cout << i << ' ' << s.substr(i) << '\n'; // 5 3 1 0 4 2
}
```

2.14 Aho-Corasick

아호-코라식(Aho-Corasick) 알고리즘은 여러 패턴 문자열을 한 번에 효율적으로 찾을 수 있게 해주는 대표적인 다중 문자열 탐색 알고리즘이다. 이 알고리즘은 먼저 트라이(Trie)를 만들어 패턴들을 저장하고, 각 노드에 실패 링크(failure link)를 추가하는데, 이 실패 링크의 원리는 KMP 알고리즘의 실패 함수와 유사하다. 이를 통해 패턴이 불일치할 때 빠르게 다음 후보로 이동할 수 있다. 탐색 과정에서는 텍스트의 각 문자를 한 번씩만 읽으면서도, 동시에 모든 패턴의 일치 여부를 판별할 수 있다. 전체 시간 복잡도는 패턴 입력 및 실패 링크 구성에 $O(\sum |P_i|)$, 텍스트 탐색에 $O(|T| + k)$가 소요된다. 여기서 $\sum |P_i|$ 는 모든 패턴의 총 길이, $|T|$는 텍스트 길이, k는 패턴의 총 등장 횟수이다.

- 아호-코라식 알고리즘은 KMP의 실패 함수 아이디어를 트라이 구조로 확장해 여러 패턴을 동시에 처리할 수 있게 했다는 점에서 의의가 있다.
- 악성코드 탐지, 금칙어 필터, 유전체 분석 등 대용량 텍스트 내 다중 패턴 검색에 널리 활용된다.
- C++ Skeleton Code (Pure)

```cpp
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;

const int ALPHABET = 26;
```

```
6   int c2i(char c) { return c - 'a'; }
7
8   // Aho-Corasick
9   struct TrieNode {
10    TrieNode* child[ALPHABET];
11    TrieNode* fail;
12    vector<int> output;
13
14    TrieNode() : fail(0) {
15      memset(child, 0, sizeof(child));
16    }
17
18    ~TrieNode() {
19      for (int i = 0; i < ALPHABET; ++i)
20      if (child[i] && child[i] != this)
21      delete child[i];
22    }
23
24    void insert(const string& word, int idx) {
25      TrieNode* cur = this;
26      for (char c : word) {
27        int nxt = c2i(c);
28        if (!cur->child[nxt]) cur->child[nxt] = new TrieNode();
29        cur = cur->child[nxt];
30      }
31      cur->output.push_back(idx);
32    }
33  };
34
35  void computeFailFunc(TrieNode* root) {
36    queue<TrieNode*> q;
37    root->fail = root;
38
39    for (int i = 0; i < ALPHABET; ++i) {
40      if (root->child[i]) {
41        root->child[i]->fail = root;
42        q.push(root->child[i]);
43      } else {
44        root->child[i] = root; // 없는 간선은 루트로 연결
45      }
46    }
47
48    while (!q.empty()) {
49      TrieNode* cur = q.front(); q.pop();
50      for (int i = 0; i < ALPHABET; ++i) {
51        TrieNode* nxt = cur->child[i];
52        if (!nxt || nxt == root) continue;
53
54        TrieNode* f = cur->fail;
55        while (!f->child[i] && f != root)
56        f = f->fail;
57        nxt->fail = f->child[i] ? f->child[i] : root;
58
59        nxt->output.insert(nxt->output.end(),
60        nxt->fail->output.begin(),
61        nxt->fail->output.end());
62
63        q.push(nxt);
64      }
65    }
66  }
67
68  //(마지막 위치, 패턴 번호) 쌍을 리턴
69  vector<pii> search(const string& text, TrieNode* root) {
70    vector<pii> res;
71    TrieNode* cur = root;
72
73    for (int i = 0; i < (int)text.size(); ++i) {
74      int idx = c2i(text[i]);
75
```

```
        while (!cur->child[idx] && cur != root)
        cur = cur->fail;
        cur = cur->child[idx];
        if (!cur) cur = root;

        for (int pat : cur->output)
        res.emplace_back(i, pat);
      }
      return res;
    }

    int main() {
      vector<string> patterns = {"he", "she", "his", "hers"};
      string text = "ushers";

      TrieNode* root = new TrieNode();
      for (int i = 0; i < patterns.size(); ++i)
      root->insert(patterns[i], i);

      computeFailFunc(root);
      auto result = search(text, root);

      cout << "Text: " << text << '\n';
      for (auto [pos, pat] : result) {
        int start = pos - (int)patterns[pat].size() + 1;
        cout << "Pattern [" << patterns[pat] << "] found at: " << start << '\n';
      }
      // Pattern [she] at: 1
      // Pattern [he] at: 2
      // Pattern [hers] at: 2
    }
```

2.15 Sqrt decomposition

제곱근 분할(sqrt decomposition)은 데이터를 $\sqrt{n}$ 크기의 블록으로 나누어, 구간 합이나 구간 최솟값 · 최댓값 등의 쿼리를 빠르게 처리하기 위한 알고리즘 기법이다. 각 블록에 대한 요약 정보를 미리 계산해 두고, 쿼리가 들어오면 필요한 블록만 참조하여 결과를 합산하거나 갱신한다. 이를 통해, 개별 원소를 전부 확인하지 않고도 효율적으로 쿼리에 응답할 수 있다. 일반적으로 쿼리와 업데이트가 모두 $O(\sqrt{n})$에 처리되어, 단순한 방법에 비해 성능을 크게 향상시킬 수 있다.

- C++ Skeleton Code (Pure)

```
    // JUNGOL 1726 구간의 최대값1
    // jungol.co.kr/problem/1726
    // sqrt decomposition : [s, e) , update() 함수 구현
    #include <cstdio>

    const int LM = 50005;
    const int MOD = 256; // sqrt(50000) = 223.606
    int N, Q;
    int A[LM];          //입력 데이터
    int B[MOD + 1];     //각 블록별 최대값

    inline int min(int a, int b) { return a < b ? a : b; }
    inline int max(int a, int b) { return a > b ? a : b; }

    void update(int idx, int newVal) {
      int bn = idx / MOD;                   //idx가 속한 블록 번호
      int sn = bn * MOD, en = min(sn + MOD, N); //idx가 속한 블록의 시작과 끝 인덱스
      A[idx] = B[bn] = newVal;
      for (int i = sn; i < en; ++i)         // 블록 내 최대값 갱신
      B[bn] = max(B[bn], A[i]);
    }

    int query(int s, int e) { // 구간 [s, e)에서 최대값 찾기
      int maxValue = A[s];
      // 1. 왼쪽 끝부분에서 블록 경계를 맞추기 위해 개별 비교
      while (s < e && s % MOD)
```

```
    maxValue = max(maxValue, A[s++]);

    // 2. 오른쪽 끝부분에서 블록 경계를 맞추기 위해 개별 비교
    while (s < e && e % MOD)
    maxValue = max(maxValue, A[--e]);

    // 3. 블록 단위로 최대값 비교
    for (s /= MOD, e /= MOD; s < e; s++)
    maxValue = max(maxValue, B[s]);

    return maxValue;
  }

  int main() {
    int s, e;
    scanf("%d %d", &N, &Q);
    for (int i = 0; i < N; ++i) {
      scanf("%d", A + i);
      update(i, A[i]);
    }

    for (int i = 0; i < Q; ++i) {
      scanf("%d %d", &s, &e);
      printf("%d\n", query(s - 1, e));
    }
  }
```

2.16 Plane sweeping

플레인 스위핑(plane sweeping) 알고리즘은 기하학적 문제, 특히 여러 개의 객체(예: 직사각형, 선분 등)가 주어졌을 때 이들 사이의 관계(예: 교차, 면적 계산 등)를 효율적으로 찾는 데 사용된다. 스위핑 라인(sweep line)을 가상으로 그려가며, 이 라인이 객체들을 스캔하면서 필요한 계산을 수행한다.

2.17 Maximum flow

최대 유량(maximum flow) 문제는 네트워크 플로우 이론에서 주어진 네트워크의 두 노드 사이(일반적으로 소스와 싱크라고 함)를 통해 흐를 수 있는 최대의 유량을 찾는 문제이다. 포드-풀커슨(Ford-Fulkerson) 알고리즘과 에드몬드-카프(Edmonds-Karp) 알고리즘 등이 이 문제를 해결하는 데 사용된다.

2.18 Bipartite match

이분 매칭(bipartite match) 문제는 이분 그래프에서 서로 다른 두 부분 집합의 노드를 매칭하는 최대 집합을 찾는 문제이다. 이는 예를 들어, 일자리 할당, 자원 분배 등 다양한 최적화 문제에서 중요한 역할을 한다. 홉크로프트-카프(Hopcroft-Karp) 알고리즘은 이분 매칭 문제를 효율적으로 해결하는 알고리즘 중 하나이다.

3 PS Tip

3.1 Tip

- 문제는 다음 링크를 통해 확인할 수 있다.
  - 백준: boj.kr/문제 번호
  - 정올: jungol.co.kr/problem/문제 번호
  - 알고스팟: algospot.com/judge/problem/read/문제 이름
- 채점용 서버는 일반적으로 1초에 1 5억번의 연산을 수행한다.
- 따라서 데이터가 $n = 10000 \sim 20000$개 일 때 $O(n^2)$ 알고리즘은 제한시간 1초 내 통과하기 어렵다.
- `#include <bits/stdc++.h>` 를 사용한다. Mac이나 Windows는 기본적으로 해당 파일이 존재하지 않으므로 해당 링크에서 다운로드 받은 후 아래 경로에 넣어준다.

- Mac: `/Library/Developer/CommandLineTools/usr/bits/stdc++.h`
- Windows: `C:\Program Files (x86)\Microsoft Visual Studio\Version\Community\VC\Tools\MSVC\Version\include\bits\stdc++.h`
- Ubuntu: `/usr/include/x86_64-linux-gnu/c++/11/bits/stdc++.h` (기본적으로 파일이 존재한다)

- 전역변수와 static 변수는 자동으로 0으로 초기화되는 반면 지역변수는 자동으로 초기화되지 않고 쓰레기 값(garbage value)으로 채워지게 된다.

```cpp
int A;        // 자동으로 0으로 초기화
int B[50];    // 자동으로 0으로 초기화
static int C; // 자동으로 0으로 초기화

int main() {
  int D;      // 쓰레기 값(garbage value)으로 채워짐
  int E[50];  // 쓰레기 값(garbage value)으로 채워짐
}
```

- $\pi = 3.14159\cdots$는 다음과 같이 여러 방법으로 사용할 수 있다

```cpp
#include <cmath>
#include <cstdio>

const double pi = acos(-1);
const double pi_ = 2.0 * acos(0);
const double pi__ = 4.0 * atan(1);

int main() {
  // 모두 <cmath> 헤더 필요
  printf("%lf\n", M_PI); // #1(GNU ok, MSVC는 #define _USE_MATH_DEFINES선언 후 사용)
  printf("%lf\n", pi);   // #2
  printf("%lf\n", pi_);  // #3
  printf("%lf\n", pi__); // #4
}
```

3.2 Time complexity

시간 복잡도는 알고리즘이 문제를 해결하는 데 걸리는 시간의 양을 나타내는 척도이다. 이는 주로 입력 크기의 함수로 표현되며, 알고리즘이 실행될 때 필요한 기본 연산의 횟수로 측정된다. 시간 복잡도를 평가할 때는 최악의 경우, 평균 경우, 최선의 경우를 고려할 수 있으나, 대부분의 경우 최악의 시간 복잡도가 중요한 지표로 사용된다. 시간 복잡도는 Big O 표기법을 사용하여 표현한다. $\log n$은 밑이 2인 $\log_2 n$을 의미한다.

- $O(1)$: 상수 시간. 입력 크기와 상관없이 알고리즘의 실행 시간이 일정하다.
- $O(\log n)$: 로그 시간. 데이터 양이 증가해도 실행 시간이 비교적 적게 증가한다. e.g., 이진 탐색
- $O(n)$: 선형 시간. 알고리즘의 실행 시간이 입력 크기에 직접 비례한다. 예를 들어, e.g., 배열 순회
- $O(n \log n)$: 선형 로그 시간. 많은 효율적인 정렬 알고리즘들이 이 시간 복잡도를 갖는다.
- $O(n^2)$: 제곱 시간. 입력 크기의 제곱에 비례하는 실행 시간을 가지며, 이중 반복문을 사용하는 알고리즘에서 흔히 발생한다.
- $O(2^n)$: 지수 시간. 알고리즘의 실행 시간이 입력 크기의 지수 함수로 증가한다. 일부 재귀 알고리즘에서 발생할 수 있다.
- $O(n!)$: 팩토리얼 시간. 알고리즘의 실행 시간이 입력 크기의 팩토리얼에 비례하여 증가한다. 순열을 생성하는 알고리즘 등에서 나타날 수 있다.

3.3 Space complexity

공간 복잡도는 알고리즘의 실행 과정에서 필요한 저장 공간의 양을 나타내는 척도이다. 이는 알고리즘이 작동하기 위해 필요한 메모리 양을 의미하며, 입력 크기와 알고리즘의 구현 방식에 따라 달라진다. 공간 복잡도 역시 Big O 표기법을 사용하여 표현되며, 주로 입력 데이터, 추가적으로 필요한 변수, 재귀 호출 시 스택 공간 등을 고려하여 계산한다.

알고리즘 설계 시, 시간 복잡도와 공간 복잡도 사이에는 종종 trade-off 관계가 있다. 예를 들어, 더 많은 메모리를 사용하여 실행 시간을 단축시키는 경우(공간을 시간으로 바꾸는 경우)가 있을 수 있다. 효율적인 알고리즘을 설계하기 위해서는 문제의 요구 사항에 따라 시간과 공간 복잡도 사이의 최적의 균형을 찾는 것이 중요하다.

3.4 Snippet code

```cpp
#include <bits/stdc++.h>
#define rnt register int
#define FASTIO cin.tie(0); cout.tie(0); ios_base::sync_with_stdio(0);
#define WATCH(x) cout << #x << " : " << x << '\n';
#define PRINT(x) cout << x << '\n';
#define X first
#define Y second
using pii = std::pair<int,int>;
using ll = long long;
using namespace std;

int main(int argc, char **argv){

}
```

vscode용 스니펫 코드는 다음과 같다. (Ctrl+Shift+P –> Snippets: Configure Snippets –> cpp.json)

```cpp
"ps": {
  "prefix": "ps",
  "body": [
  "#include <bits/stdc++.h>",
  "#define rnt register int",
  "#define FASTIO cin.tie(0); cout.tie(0); ios_base::sync_with_stdio(0);",
  "#define WATCH(x) cout << #x << \": \" << x << '\\n';",
  "#define PRINT(x) cout << x << '\\n';",
  "#define X first",
  "#define Y second",
  "using pii = std::pair<int,int>;",
  "using ll = long long;",
  "using namespace std;",
  "",
  "int main(int argc, char **argv){",
    "  $1",
    "}"
  ],
  "description": ""
}
```

3.5 Macro

1. C++11부터 define 대신 constexpr을 사용할 수 있다.
   - `#define LM 10005`
   - `constexpr int LM = 10005;`
2. `register int`는 int 자료형보다 미세하게 실행 속도가 빠르다. 이는 constexpr 로는 선언이 안되므로 항상 `#define` 으로 선언해줘야 한다.
   - `#define rnt register int`

3. `#define POWER(x)x*x` 과 같이 선언하면 안된다. `POWER(2+3)= 2+3*2+3 = 11`이 되므로 `#define POWER(x)((x)*(x))` 와 같이 항상 괄호로 감싸줘야 한다.

    - `#define POWER(x)((x)*(x))`
    - `#define ABS(x)((x)>0 ? (x):-(x))`
    - `#define MIN(x,y)((x)<(y)? (x): (y))`
    - `#define MAX(x,y)((x)<(y)? (y): (x))`
    - `#define ALL(x)begin(x), end(x)`
    - `#define SIZE(x)int(size(x))`
    - `#define REP(i,a,b)for(int i=(a); i<=(b); i++)`

4. 디버깅용 매크로

    - `#define WATCH(x)cout << #x << " : " << x << '\n'`
    - `#define DEBUG(x)cout << "DB - " << x << '\n'`
    - `#define PRINT(x)cout << x << '\n'`
    - `#define NWATCH(x)`
    - `#define NDEBUG(x)`
    - `#define NPRINT(x)`
        - DEBUG, WATCH, PRINT 앞에 N만 입력해주면 출력 구문을 비활성화할 수 있다.

3.6 Value initialization

`memset()` 함수는 배열을 0 또는 −1 값으로 초기화할 때만 사용한다. 이외의 값은 `memset` 함수로 세팅할 수 없다. 배열을 특정 값으로 세팅하고 싶다면 단순하게 for 문을 돌리거나 algorithm 헤더의 `std::fill()` 함수를 사용하면 된다.

- `memset(A, 0, sizeof(A));`
- `for(int i=0; i<N; i++)A[i] = k;` : k는 임의의 수
- `std::fill(A, A+N, k);`

예외적으로 다익스트라, 플로이드 와샬 알고리즘 등 그래프 최단거리 문제 등에서 무한대 값을 나타낼 때 `memset()`이 종종 사용되기도 한다

- `memset(A, 0x3f, sizeof(A));`

이는 A의 값을 `0x3f3f3f3f (=1,061,109,567)`으로 변경하여 해당 배열이 무한대(INF)로 초기화되었음을 의미한다. `0x3f3f3f3f`은 2배를 해도 `int` 범위 내에 있기 때문에 오버플로우로부터 안전하여 무한대 값으로 자주 사용된다

```
1  constexpr int INF = 0x3f3f3f3f; // 무한대 값으로 자주 사용됨
```

3.7 I/O stream

1. cin/cout을 사용할 경우 다음 명령을 실행시켜야 시간 초과가 발생하지 않는다

    - `ios_base::sync_with_stdio(0);` : C stream과 C++ stream의 동기화를 비활성화하여 C++ stream만 사용. 속도 측면에서 이득이지만 이후부터는 `printf`와 `cout`을 섞어쓰면 안된다.
    - `cin.tie(0), cout.tie(0)` : `cin, cout` 명령 수행 전 `flush` 버퍼를 비우는 과정을 비활성화함으로써 속도 측면에서 이득을 봄 (하지만 입출력의 순서를 보장받을 수 없기 때문에 PS 이외에는 권장하는 방법은 아님)
    - `cin.tie(0)->sync_with_stdio(0);` : 둘을 한 줄로 작성 가능

2. `std::endl`은 개행 문자 "를 입력 후 버퍼를 비우는 과정을 자동으로 수행하기 때문에 느리므로 사용하지 않는다. `endl` 대신 "을 사용한다.

3. `getline(cin, s)` : 공백까지 통째로 한 줄 입력을 받을 때 사용

4. `cout << fixed << setprecision(n)` : 소수점 아래 n자리까지 반올림하여 출력할 때 사용

3.8 Range-based for loop

1. C++11부터 추가된 기능

   - `for(int v : V)` : 벡터 V의 모든 원소의 복사본 v대해 루프를 돈다
   - `for(int &v : V)` : 벡터 V의 모든 원소의 원본 v에 대해 루프를 돈다

3.9 Bit-wise operation

유용하게 사용되는 비트연산자

- a를 k로 나눈 나머지 값 (k가 2의 거듭제곱인 경우만)

```
result = a & (k-1); // same as 'a % k'    (k = 2, 4, 8, 16, ... 2^n)
```

- a가 홀수인지 판단하는 법

```
if(a & 1) { printf("odd"); }
```

- a가 짝수인지 판단하는 법

```
if(~a & 1) { printf("even"); }
```

- a와 2^k을 곱한 결과

```
result = a << k;
```

- a를 2^k로 나눈 몫 p와 나머지 q

```
p = a >> k;
q = a & ((1 << k) - 1);
```

- a와 b의 값을 서로 바꾸는 코드

```
a = a^b, b = a^b, a = a^b;
```

- a가 2의 제곱수인지 판별

```
result = a & (a-1);   // a!=0 이면서 result==0이면 2의 제곱수
```

- k개의 집합 원소가 주어졌을 때

```
a = (1 << k) - 1;          // 꽉 찬 집합 구하기
a = 0;                         // 공집합 구하기
a |= (1 << k);             // k번째 원소의 추가
a &= ~(1 << k);          // k번째 원소의 삭제
a ^= (1 << k);             // k번째 원소의 토글 (0이면 1, 1이면 0으로 변경)
result = a & (1 << k); // k번째 원소의 포함 여부 확인
result = (a | b);          // a,b의 합집합
result = (a & b);         // a,b의 교집합
result = (a & ~b);       // a에서 b를 뺀 차집합
result = (a ^ b);         // a와 b 중 하나에만 포함된 원소들의 집합
```

- 최소 원소 지우기 (a가 2의 제곱수인지 판별하는 코드와 동일)

```
int main(){
  int a = 40;  // 40 (0b101000)
  // 39 (0b100111)
  a &= a-1;
  printf("%d\n", a); // 32 (0b100000)
}
```

- 집합의 크기 구하기

```
int bitCount(int x){
  if(x==0) return 0;
  return x%2 + bitCount(x/2);
}

int main() {
  int a = 72;   // 0b01001000

  printf("%d\n", bitCount(a));        // 2, Naive method
  printf("%d\n", __builtin_popcount(a)); // 2, Optimized method(gcc/g++ only)
}
```

- 모든 부분 집합 순회하기 (공집합 제외)

```
int main(){
  int a = 40; // 0b101000

  for(int s=a; s; s=((s-1) & a)) {
    printf("%d ", s); // 40(0b101000) 32(0b100000) 8(0b001000)
  }
}
```

- a의 Least Significant 1 Bit (2^0부터 시작하여 처음 만나는 1인 비트의 가중치 값)을 구하는 법

```
int main() {
  int a = 72; // 0b01001000

  int lsb = (a & -a);
  printf("a's LSB magnitude %d\n", lsb);      // 8, 2^3
  printf("a's LSB order %d\n", __builtin_ctz(a)); // 3 (gcc/g++ only)
}
```

3.10 Partial sum

1. 1D 부분합을 구하는 코드

   - $p[i] = \sum_{j=0}^{i} A[j]$
   - $sum(a, b) = p[b] - p[a-1]$

```
#include <bits/stdc++.h>
using namespace std;

vector<int> partialSum( const vector<int>& A ) {
  vector<int> ret( A.size() );
  ret[0] = A[0];
  for ( int i = 1; i < A.size(); i++ ) {
    ret[i] = ret[i - 1] + A[i];
  }
  return ret;
}

int rangeSum( const vector<int>& psum, int a, int b ) {
  if ( a == 0 ) return psum[b];
  return psum[b] - psum[a - 1];
}

```

```cpp
int main( int argc, char **argv ) {
  vector<int> A;

  A.push_back(10);  // 0
  A.push_back(9);   // 1
  A.push_back(6);   // 2 <--
  A.push_back(7);   // 3
  A.push_back(6);   // 4
  A.push_back(2);   // 5 <--
  A.push_back(4);   // 6
  A.push_back(5);   // 7
  A.push_back(1);   // 8
  A.push_back(8);   // 9

  vector<int> psum = partialSum( A );
  int ret = rangeSum( psum, 2, 5 ); // (6+7+6+2) = 21
  printf( "%d\n", ret );
}
```

2. 2D 부분합을 구하는 코드

- $p[y,x] = \sum_{i=0}^{y} \sum_{j=0}^{x} A[i,j]$
- $sum(y_1, x_1, y_2, x_2) = p[y_2, x_2] - p[y_2, x_1 - 1] - p[y_1 - 1, x_2] + p[y_1 - 1, x_1 - 1]$

```cpp
#include <bits/stdc++.h>
using namespace std;

int gridSum( const vector<vector<int>>& psum, int y1, int x1, int y2, int x2 ) {
  return psum[y2][x2] - psum[y1 - 1][x2] - psum[y2][x1 - 1] + psum[y1 - 1][x1 - 1];
}

int main() {
  int n = 8;
  vector<vector<int>> A(n, vector<int>(n));
  vector<vector<int>> psum(n, vector<int>(n));

  for (int i = 0, value = 1; i < n; i++) {
    for (int j = 0; j < n; j++, value++) {
      A[i][j] = value;

      psum[i][j] = A[i][j]
      + (i > 0 ? psum[i - 1][j] : 0)
      + (j > 0 ? psum[i][j - 1] : 0)
      - (i > 0 && j > 0 ? psum[i - 1][j - 1] : 0);
    }
  }

  // (1,1)부터 (4,4)까지의 부분합 출력
  printf("%d\n", gridSum(psum, 1, 1, 4, 4)); // 376
}
```

3. 부분합을 사용하여 분산을 구하는 코드

- $var = \frac{1}{b-a+1} \cdot \sum_{i=a}^{b} (A[i] - m)^2$
- $= \frac{1}{b-a+1} \cdot \sum_{i=a}^{b} (A[i]^2 - 2A[i] \cdot m + m^2)$
- $= \frac{1}{b-a+1} \cdot \left(\sum_{i=a}^{b} A[i]^2 - 2m \cdot \sum_{i=a}^{b} A[i] + (b-a+1)m^2 \right)$

```cpp
#include <bits/stdc++.h>
using namespace std;

vector<int> partialSum( const vector<int>& A ) {
  vector<int> ret( A.size() );
  ret[0] = A[0];
  for ( int i = 1; i < A.size(); i++ ) {
    ret[i] = ret[i - 1] + A[i];
```

```
    }
    return ret;
  }

  int rangeSum( const vector<int>& psum, int a, int b ) {
    if ( a == 0 ) return psum[b];
    return psum[b] - psum[a - 1];
  }

  double variance( const vector<int>& sqpsum,
  const vector<int>& psum, int a, int b ) {
    double tot = b - a + 1;
    double mean = rangeSum( psum, a, b ) / tot;
    double ret = rangeSum( sqpsum, a, b ) - 2 * mean * rangeSum( psum, a, b ) + tot * mean * mean;
    return ret / tot;
  }

  int main() {
    vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> psum = partialSum(data);

    vector<int> sqdata(data.size());
    for (int i = 0; i < data.size(); i++) {
      sqdata[i] = data[i] * data[i];
    }
    vector<int> sqpsum = partialSum(sqdata);

    // 구간 [a, b]의 분산 계산
    int a = 4, b = 6;
    printf("%lf\n", variance(sqpsum, psum, a, b)); // 0.6666
  }
```

3.11 Modulo operation

Mod 연산(%)은 나머지를 구하는 연산으로 다음과 같은 동치 관계가 성립한다

- `(a + b)% MOD = ((a % MOD)+ (b % MOD))% MOD`
- `(a + b + c)% MOD = ((a % MOD)+ (b % MOD)+ (c % MOD))% MOD`

3.12 Base conversion

- A진법을 10진법으로 변환하는 코드 (Horner's method 사용)
  - `result = (((0*A + digit1)* A + digit2)* A + digit3)* A + ...`
  - `A: base`

```
  int main() {
    int A;
    char S[100];
    scanf("%d %s", &A, S);

    int ret = 0;
    for(int i=0; S[i]; i++) {
      if(S[i] < 'A') ret = ret * A + (S[i] - '0');
      else           ret = ret * A + (S[i] - 'A' + 10);
    }

    printf("%d", ret);
  }
```

- 10진법을 B진법으로 변환하는 코드 (재귀 함수 사용) (한 글자씩 바로 출력)

```
  char chr[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

  void Get(int ret, int to) {
```

```
    if(ret < to) {
      printf("%c", chr[ret]);
      return;
    }
    Get(ret/to, to);
    printf("%c", chr[ret%to]);
  }

  int main() {
    int ret, B;
    scanf("%d %d", &ret, &B);

    Get(ret, B);
  }
```

3.13 std::string

- 문자열을 반복 연산할 때 흔히하는 실수 (L은 문자열, $|L|$은 문자열 길이)
  - `for ( int i = 0; i < 1000000; i++ ){ s = s + 'a'; }` : 시간복잡도 $O(|L|^2)$
  - `for ( int i = 0; i < 1000000; i++ ){ s += 'a'; }` : 시간복잡도 $O(|L|)$

```
#include <bits/stdc++.h>
using namespace std;

int main( int argc, char **argv ) {
  string s;

  // 시간복잡도: O(N^2)
  // s+'a'라는 객체를 새로 만든 후 이를 s에 대입
  for ( int i = 0; i < 1'000'000; i++ ) {
    s = s + 'a';
  }

  // 시간복잡도: O(N)
  // += 연산자를 이용하면 시간복잡도가 더해지는 길이에만 영향 받음
  for ( int i = 0; i < 1'000'000; i++ ) {
    s += 'a';
  }
}
```

- 문자열을 특정 구분자(seperator, delimeter) 단위로 나눈 결과를 반환하는 split 함수

```
#include <bits/stdc++.h>
using namespace std;

vector<string> split( string& s, string& sep ) {
  vector<string> ret;
  int pos = 0;
  while ( pos < s.size() ) {
    int nxt_pos = s.find( sep, pos );
    if ( nxt_pos == -1 ) nxt_pos = s.size();
    if ( nxt_pos - pos > 0 )
    ret.push_back( s.substr( pos, nxt_pos - pos ) );
    pos = nxt_pos + sep.size();
  }
  return ret;
}

int main( int argc, char **argv ) {
  string s = "aaaaa,bbbbb ccccc,ddddd.eeeee";
  string sep1 = " ";
  string sep2 = ",";
  string sep3 = ".";
  vector<string> sp = split( s, sep2 );

  for ( auto s : sp )
```

```
25        cout << s << endl;
26    }
```

4 Problem list

4.1 Data structure

4.1.1 Array

- **Level1 – 개념 문제**
  - ALGOSPOT 조세푸스 문제 (JOSEPHUS), Solution

4.1.2 Linked List

- **Level1 – 개념 문제**
  - BOJ 1406 에디터, Solution1, Solution2

4.1.3 Stack

- **Level1 – 개념 문제**
  - BOJ 10828 스택, Solution BOJ 9093 단어 뒤집기, Solution BOJ 17298 오큰수, Solution BOJ 4949 균형잡힌 세상, Solution1, Solution2 ALGOSPOT 짝이 맞지 않는 괄호 (BRACKETS2), Solution

4.1.4 Queue

- **Level1 – 개념 문제**
  - BOJ 10845 큐, Solution ALGOSPOT 외계 신호 분석 (ITES), Solution

4.1.5 Deque

- **Level1 – 개념 문제**
  - BOJ 10866 덱, Solution

4.1.6 Priority Queue

- **Level1 – 개념 문제**
  - BOJ 11286 절대값 힙, Solution BOJ 1715 카드 정렬하기, Solution ALGOSPOT 변화하는 중간값 (RUNNINGMEDIAN), Solution

4.1.7 Tree

- **Level1 – 개념 문제**
  - BOJ 1991 트리순회, Solution BOJ 11725 트리의 부모 찾기, Solution ALGOSPOT 트리 순회 순서 변경 (TRAVERSAL), Solution
- **Level2 – 발전 문제**
  - BOJ 2250 트리의높이와너비, Solution ALGOSPOT 요새(FORTRESS), Solution

4.1.8 Binary Search Tree (set, map)

- **Level1 – 개념 문제**
  - BOJ 7662 이중 우선순위 큐, Solution BOJ 1202 보석 도둑, Solution ALGOSPOT 삽입 정렬 뒤집기 (INSERTION), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 너드인가, 너드가 아닌가2 (NERD2), Solution

4.1.9 Hash

- **Level1 – 개념 문제**
  - BOJ 7785 회사에 있는 사람, Solution BOJ 1620 나는야 포켓몬 마스터 이다솜, Solution JUNGOL 4853 문자열 별칭, Solution JUNGOL 4854 땅 점령 게임, Solution
- **Level2 – 발전 문제**
  - JUNGOL 1155 MCS, Solution

4.1.10 Union-find

- **Level1 – 개념 문제**
  - BOJ 1717 집합의 표현, Solution BOJ 7511 소셜 네트워킹 어플리케이션, Solution JUNGOL 1863 종교, Solution1, Solution2 ALGOSPOT 에디터 전쟁(EDITORWARS), Solution
- **Level2 – 발전 문제**
  - BOJ 13306 트리, Solution BOJ 2463 비용, Solution

4.1.11 Trie

- **Level1 – 개념 문제**
  - BOJ 14425 문자열 집합, Solution1, Solution2 ALGOSPOT 안녕히 그리고 물고기는 고마웠어요 (SOLONG), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 보안종결자 (NH), Solution (Aho-Corasick)

4.2 Algorithm

4.2.1 Basic I/O & Simulation

- **Level1 – 개념 문제**
  - BOJ 2557 Hello World, Solution BOJ 11718 그대로 출력하기, Solution1, Solution2 BOJ 1000 A+B, Solution BOJ 11720 숫자의 합, Solution BOJ 10953 A+B-6, Solution BOJ 2741 N찍기, Solution BOJ 2739 구구단, Solution BOJ 1924 2007년, Solution BOJ 10818 최소, 최대, Solution BOJ 4101 크냐?, Solution BOJ 10871 X보다 작은 수, Solution BOJ 10808 알파벳 개수, Solution BOJ 11723 집합, Solution1, Solution2 BOJ 17413 단어 뒤집기2, Solution BOJ 11655 ROT13, Solution BOJ 10824 네 수, Solution JUNGOL 3574 수식계산기1, Solution JUNGOL 1105 수식계산기2, Solution
- **Level2 – 발전 문제**
  - BOJ 2658 직각이등변삼각형찾기, Solution JUNGOL 1374 긴 자리 덧셈 뺄셈, Solution1, Solution2, Solution3, Solution4

4.2.2 String

- **Level1 – 개념 문제**
  - BOJ 1543 문서 검색, Solution BOJ 2941 크로아티아 알파벳, Solution

4.2.3 Bit-wise operation

- **Level1 – 개념 문제**
  - BOJ 14391 종이조각, Solution BOJ 11576 Base Conversion, Solution BOJ 11723 집합, Solution3 BOJ 1497 기타콘서트, Solution JUNGOL 1419 엔디안, Solution JUNGOL 3293 비트연산 1, Solution JUNGOL 1239 비밀편지, Solution JUNGOL 3112 보안시스템, Solution1
- **Level2 – 발전 문제**
  - ALGOSPOT 졸업 학기 (GRADUATION), Solution

4.2.4 Brute Force

- **Level1 – 개념 문제**
  - BOJ 18111 마인크래프트, Solution BOJ 2309 일곱난쟁이, Solution BOJ 3085 사탕게임, Solution ALGOSPOT 게임판 덮기 (BOARDCOVER), Solution
- **Level2 – 발전 문제**
  - BOJ 2642 전개도, Solution ALGOSPOT 시계 맞추기 (CLOCKSYNC), Solution

4.2.5 Math

- **Level1 – 개념 문제**
  - BOJ 7571 점 모으기, Solution BOJ 10158 개미, Solution BOJ 2477 참외밭, Solution BOJ 2527 직사각형, Solution BOJ 2503 숫자야구, Solution BOJ 10430 나머지, Solution BOJ 10972 다음순열, Solution1, Solution2 BOJ 10973 이전순열, Solution BOJ 10974 모든순열, Solution BOJ 10819 차이를최대로, Solution BOJ 10971 외판원순회2, Solution BOJ 11005 진법변환2, Solution BOJ 2745 진법변환, Solution BOJ 1373 2진수8진수, Solution BOJ 1212 8진수2진수, Solution BOJ 10827 팩토리얼, Solution BOJ 1978 소수 찾기, Solution BOJ 1929 소수구하기, Solution BOJ 2609 최대공약수와 최소공배수, Solution BOJ 9613 GCD 합, Solution BOJ 6603 로또, Solution BOJ 11653 소인수분해, Solution BOJ 11050 이항계수1, Solution BOJ 11051 이항계수2, Solution JUNGOL 3106 진법변환, Solution JUNGOL 3135 const 구간의 합 구하기 1D, Solution JUNGOL 3136 const 구간의 합 구하기 2D, Solution
- **Level2 – 발전 문제**
  - BOJ 6064 카잉 달력, Solution JUNGOL 4692 눈 내리는 겨울밤, Solution JUNGOL 3109 숫자야구2, Solution ALGOSPOT 비밀번호 486 (PASS486), Solution1, Solution2 ALGOSPOT 마법의 약 (POTION), Solution1, Solution2 ALGOSPOT 크리스마스 인형 (CHRISTMAS), Solution

4.2.6 Recursion

- **Level1 – 개념 문제**
  - BOJ 1629 곱셈, Solution BOJ 11729 하노이 탑 이동 순서, Solution

4.2.7 Backtracking

- **Level1 – 개념 문제**
  - BOJ 1182 부분수열의 합, Solution BOJ 13023 ABCDE, Solution BOJ 15649 N과M (1), Solution BOJ 9663 N-Queen, Solution JUNGOL 1169 주사위 던저기1, Solution JUNGOL 1127 맛있는 음식 (PERKET), Solution ALGOSPOT 피크닉 (PICNIC), Solution

4.2.8 Two pointers

- **Level1 – 개념 문제**
  - BOJ 2467 용액, Solution BOJ 2230 수 고르기, Solution BOJ 1806 부분 합, Solution

4.2.9 DFS

- **Level1 – 개념 문제**
  - BOJ 11724 연결요소의 개수, Solution BOJ 1707 이분그래프, Solution ALGOSPOT 단어 제한 끝말잇기 (WORDCHAIN), Solution ALGOSPOT 고대어 사전 (DICTIONARY), Solution ALGOSPOT 감시 카메라 설치 (GALLERY), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 회의실 배정 (MEETINGROOM), Solution

4.2.10 BFS

- **Level1 – 개념 문제**
  - BOJ 2178 미로탐색, Solution1, Solution2 BOJ 4963 섬의 개수, Solution BOJ 1260 DFS와 BFS, Solution BOJ 14501 퇴사, Solution1, Solution2 BOJ 7576 토마토, Solution BOJ 1926 그림, Solution JUNGOL 1078 저글링방사능오염, Solution JUNGOL 1462 보물섬, Solution JUNGOL 2606 토마토 (초), Solution JUNGOL 3640 잡기 놀이 (catch the cow), Solution JUNGOL 2261 경로 찾기, Solution ALGOSPOT Sorting Game (SORTGAME), Solution ALGOSPOT 하노이의 네 탑 (HANOI4), Solution
- **Level2 – 발전 문제**
  - BOJ 2452 그리드 게임, Solution BOJ 4179 불!, Solution BOJ 1697 숨바꼭질, Solution JUNGOL 2058 고돌이 고소미, Solution ALGOSPOT 어린이날 (CHILDRENDAY), Solution

4.2.11 Binary search

- **Level1 – 개념 문제**
  - BOJ 1920 수 찾기, Solution JUNGOL 3517 이진탐색, Solution ALGOSPOT 승률 올리기 (RATIO), Solution
- **Level2 – 발전 문제**
  - BOJ 10816 숫자 카드2, Solution1, Solution2 BOJ 18870 좌표 압축, Solution1, Solution2 BOJ 2295 세 수의 합, Solution

4.2.12 Ternary search

- **Level2 – 발전 문제**
  - ALGOSPOT 꽃가루 화석 (FOSSIL), Solution

4.2.13 Parametric search

- **Level1 – 개념 문제**
  - BOJ 1654 랜선 자르기, Solution ALGOSPOT 남극 기지 (ARCTIC), Solution ALGOSPOT DARPA Grand Challenge (DARPA), Solution ALGOSPOT 캐나다 여행 (CANADATRIP), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 수강 철회 (WITHDRAWAL), Solution

4.2.14 Sort

- **Level1 – 개념 문제**
  - BOJ 2751 수 정렬하기 2, Solution BOJ 15688 수 정렬하기 5, Solution JUNGOL 3339 계수정렬 (Counting Sort), Solution JUNGOL 2082 힙정렬2, Solution JUNGOL 3519 병합정렬, Solution
- **Level2 – 발전 문제**
  - BOJ 7453 합이 0인 네 정수, Solution BOJ 11652 카드, Solution JUNGOL 3120 기수정렬 (Radix Sort), Solution

4.2.15 Topological sort

- **Level1 – 개념 문제**
  - BOJ 2252 줄 세우기, Solution

4.2.16 Dynamic programming

- **Level1 – 개념 문제**
  - BOJ 2193 이친수, Solution BOJ 9095 123더하기, Solution BOJ 1463 1로 만들기, Solution BOJ 12852 1로 만들기 2, Solution BOJ 2579 계단 오르기, Solution BOJ 1149 RGB거리, Solution BOJ 11726 2xN 타일링, Solution BOJ 11649 구간 합 구하기 4, Solution JUNGOL 1411 두 줄로 타일깔기, Solution JUNGOL 1520 계단 오르기, Solution JUNGOL 2000 동전 교환, Solution JUNGOL 3112 보안시스템, Solution ALGOSPOT 외발 뛰기 (JUMPGAME), Solution ALGOSPOT 삼각형 위의 최대 경로 (TRIANGLEPATH), Solution ALGOSPOT 최대 부분 증가 수열 (LIS), Solution ALGOSPOT 합친 LIS (JLIS), Solution ALGOSPOT 원주율 외우기 (PI), Solution ALGOSPOT 타일링 (TILING2), Solution ALGOSPOT 달팽이 (SNAIL), Solution ALGOSPOT 비대칭 타일링 (ASYMTILING), Solution1, Solution2
- **Level2 – 발전 문제**
  - BOJ 15990 123더하기 5, Solution JUNGOL 2063 택배, Solution ALGOSPOT 와일드카드 (WILDCARD), Solution ALGOSPOT 양자화 (QUANTIZE), Solution ALGOSPOT 삼각형 위의 최대 경로 개수 세기 (TRIPATHCNT), Solution ALGOSPOT 폴로오미노 (POLY), Solution ALGOSPOT 두니발 박사의 탈옥 (NUMB3R), Solution1, Solution2

4.2.17 Greedy

- **Level1 – 개념 문제**
  - BOJ 11047 동전0, Solution BOJ 1931 회의실 배정, Solution BOJ 2217 로프, Solution BOJ 1026 보물, Solution JUNGOL 1816 외양간, Solution ALGOSPOT 출전 순서 정하기 (MATCHORDER), Solution ALGOSPOT 도시락 데우기 (LUNCHBOX), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 문자열 합치기 (STRJOIN), Solution ALGOSPOT 미나스 아노르 (MINASTIRITH), Solution

4.2.18 Divide and conquer

- **Level1 – 개념 문제**
  - ALGOSPOT 쿼드 트리 뒤집기 (QUADTREE), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 울타리 잘라내기 (FENCE), Solution

4.2.19 Sqrt decomposition

- **Level1 – 개념 문제**
  - JUNGOL 1726 구간의 최대값1, Solution1 JUNGOL 2469 줄세우기, Solution1

4.2.20 Segment tree

- **Level1 – 개념 문제**
  - JUNGOL 1726 구간의 최대값1, Solution2, Solution3 JUNGOL 2469 줄세우기, Solution2
  - ALGOSPOT 등산로 (MORDOR), Solution ALGOSPOT 삽입 정렬 시간 재기 (MEASURETIME), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 족보 탐험 (FAMILYTREE), Solution

4.2.21 Minimum spanning tree (Prim, Kruskal)

- **Level1 – 개념 문제**
  - BOJ 1197 최소 스패닝 트리, Solution1, Solution2 ALGOSPOT 근거리 네트워크 (LAN), Solution
- **Level2 – 발전 문제**
  - BOJ 1368 물대기, Solution ALGOSPOT 여행 경로 정하기 (TPATH), Solution

4.2.22 Floyd-Warshall

- **Level1 – 개념 문제**
  - BOJ 11404 플로이드, Solution BOJ 11780 플로이드2, Solution
  - ALGOSPOT 음주 운전 단속 (DRUNKEN), Solution ALGOSPOT 선거 공약 (PROMISES), Solution

4.2.23 Dijkstra

- **Level1 – 개념 문제**
  - BOJ 1753 최단경로, Solution BOJ 11779 최소비용 구하기2, Solution
  - ALGOSPOT 신호 라우팅 (ROUTING), Solution ALGOSPOT 소방차 (FIRETRUCKS), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 철인 N종 경기 (NTHLON), Solution

4.2.24 Bellman-Ford

- **Level1 – 개념 문제**
  - ALGOSPOT 시간 여행 (TIMETRIP), Solution

4.2.25 KMP

- **Level1 – 개념 문제**
  - BOJ 16916 부분 문자열, Solution ALGOSPOT 작명하기 (NAMING), Solution ALGOSPOT 재하의 금고 (JAEHASAFE), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 팰린드롬 만들기 (PALINDROMIZE), Solution

4.2.26 Suffix array (Manber-Myers)

- **Level1 – 개념 문제**
  - ALGOSPOT 말버릇 (HABIT), Solution

4.2.27 Network flow (Ford-Fulkerson)

- **Level1 – 개념 문제**
  - ALGOSPOT 승부 조작 (MATCHFIX), Solution

4.2.28 Bipartite matching

- **Level1 – 개념 문제**
  - ALGOSPOT 비숍 (BISHOPS), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 함정 설치 (TRAPCARD), Solution

4.2.29 Computational geometry

- **Level1 – 개념 문제**
  - ALGOSPOT 너드인가 너드가 아닌가 (NERDS), Solution
- **Level2 – 발전 문제**
  - ALGOSPOT 핀볼 시뮬레이션 (PINBALL), Solution ALGOSPOT 보물섬 (TREASURE), Solution

5 Reference

[1] (Blog) 알고리즘 문제풀이(PS) 시작하기 - plzrun

[2] (Blog) 내가 문제풀이를 연습하는 방법 - koosaga

[3] (Blog) 코드포스 블루, 퍼플 달성 후기 및 일지 공부법 - Rebro

[4] (Blog) PS를 공부하는 방법 (How to study Problem Solving?) - subinium

[5] (Blog) BaaaaaaaarkingDog의 강좌/실전 알고리즘

[6] (Youtube) BaaaaaaaarkingDog의 강좌/실전 알고리즘

[7] (Site) C++ 유용한 기능들

[8] (Book) 알고리즘 문제 해결 전략 - 구종만 저

6 Revision log

- 1st: 2025-06-28